Tomi Tuhkanen

Software Platform Architecture for Laboratory Workstation Software

Helsinki Metropolia University of Applied Sciences

Master of Engineering

Information Technology

Master's Thesis

15 April 2013

Helsinki Metropolia University of Applied Sciences

Author(s) Title Number of Pages Date	Tomi Tuhkanen Software Platform Architecture for Laboratory Workstation Software 69 pages 15 April 2013
Degree	Master of Engineering
Degree Programme	Information Technology
Specialisation option	Mobile Programming
Instructor(s)	Peeter Kitsnik, PhD, Senior Lecturer Mika Salkola, M.Sc. (Eng.), R&D Manager

The aim of the thesis was to design an architecture for a workstation software platform to control laboratory instruments and to decide technologies that were used with the platform. The platform needed to support multiple instruments with different functionalities and also support multiple simultaneous instruments. The application, based on the architecture, needed to function as a stand-alone application and as an automation service a. The platform was to be developed with Microsoft .NET.

The project was started with a study of existing architectures and design guides. An aim was to find some common factors and enterprise design patterns that would help in designing a preliminary architecture. Brief technological studies were carried out to decide the technologies to be used with the software platform.

Based on the study, the first draft of the architecture was designed and the implementation of the platform was started with the chosen technologies. The architecture design and implementation was done in an iterative manner, so the design kept evolving through the first months of the implementation as specifications and the domain knowledge increased.

As a result, modular and decoupled architecture seemed to fit well for the foundation of this kind of application. It gives enough flexibility, so the design can provide functionality for different use cases, and changes to design only affect small parts of architecture. This kind of architecture allows the application to be used in multiple different ways, for example as a centralized service or as a stand-alone application, and required modules can be used with an automation service.

Keywords	software architecture, modularity, design patterns, .NET technologies

Työn tekijä Työn nimi Sivumäärä Päivämäärä	Tomi Tuhkanen Ohjelmistoalusta-arkkitehtuuri laboratioriotyöasemaohjelmistolle 69 sivua 15.4.2013
Tutkinto	Master of Engineering
Koulutusohjelma	Information Technology
Suuntautumisvaihtoehto	Mobile Programming
Ohjaajat	Peeter Kitsnik, PhD, Senior Lecturer Mika Salkola, M.Sc. (Eng.) R&D Manager

Työn päämääränä oli suunnitella ohjelmistoalusta-arkkitehtuuri, joka palvelee laboratoriomittalaitteita, sekä valita ohjelmistoalustassa käytettävät teknologiat. Ohjemiston pitää tukea monia erilaisia mittalaitteita, joilla on erilaisia toimintoja, sekä tukea monia mittalaitteita samanaikaisesti. Ohjelmiston pitää toimia itsenäisenä ohjelmana sekä automaatiopalveluna. Ohjelmistoalusta kehitetään Microsoft .NET Frameworkilla.

Projekti aloitettiin tutkimalla olemassaolevia ratkaisumalleja ja suunnitteluohjeita. Tarkoituksena oli löytää yhteisiä tekijöitä ja malleja, jotka auttavat alustavan arkkitehtuurin suunnittelussa. Projektin alussa tehtiin myös teknologiatutkimuksia, joiden avulla päätettiin käytettävät teknologiat.

Tutkimusten perusteella suunniteltiin ensimmäinen versio arkkitehtuurista ja aloitettiin ohjelmistoalustan toteutus. Arkkitehtuurin suunnitelma ja toteutus tehtiin iteratiivisella mallilla, joten suunnitelma muuttui ensimmäisten kuukausien aikana huomattavasti, kun spesifikaatiot tarkentuivat ja sovellusalueen tuntemus lisääntyi.

Lopputuloksena modulaarinen ja löyhäkytketty arkkitehtuuri vaikuttaa sopivan hyvin ohjelmistolle, jonka pitää muokkautua eri käyttötapauksiin. Se antaa tarpeeksi joustavuutta, jolloin arkkitehtuuri voi tuoda vaaditut toiminnallisuudet kaikille käyttötapauksille ja muutokset arkkitehtuuriin vaativat muutoksia vain pienempiin osiin ohjelmistossa. Modulaarinen ja löyhäkytketty arkkitehtuuri sallii ohjelmiston käytön eri käyttötapauksissa, esimerkiksi keskitettynä palveluna, itsenäisenä ohjelmana ja automaatiopalveluna.

Avainsanat	ohjelmistoarkkitehtuuri, modulaarisuus, suunnittelumallit, .NET teknologiat

Contents

Abbreviations

CI	Continuous Integration
CLR	Common Language Runtime
DI	Dependency Injection
EF	Entity Framework
GUID	Globally Unique Identifier
MEF	Managed Extensibility Framework
MVVM	Model-View-ViewModel
POCO	Plain Old CLR Object
SQL	Structured Query Language
UT	Unit Test
WCF	Windows Communication Foundation
WPF	Windows Presentation Framework
XAML	Extensible Application Markup Language
XML	Extensible Markup Language

1 Introduction

The software architecture of an application is the structure of the system, including systems decomposition into parts. The architecture defines decomposed parts connectivity and interaction mechanisms, and the guiding decisions and principles that were used in the design of the system.

Software, like any other complex structure, must be built on a solid foundation. Failing to consider key scenarios and requirements, common problems and to prepare for future changes can put the application at risk. Poor architecture may lead to software that is not able to support existing or future requirements or to software that does not have the required performance or is unstable.

To help building foundations and unifying development, companies develop software platforms. A software platform is a set of subsystems, which form a common infrastructure. Related software applications are developed on top of that infrastructure. Platforms enable faster delivery of products and updates. Furthermore they improve the quality of the software and reduce the common risks that are always associated with developing software. However, developing a platform has also a downside. The initial development of the platform requires more time, than just releasing an application without a platform. Platform design also requires design skills and knowledge of the domain. Even after the initial release it requires skill to keep the architecture structure up to date.

In this project, I will first identify the key concepts based on common software goals and qualities. The key concepts try to tackle common problems that have arisen with modern software development. Some of the concepts are related to software, such as reliability and maintainability, and some to external risks, such as frequent changes in development resources. These key concepts then define what is required from the application and are used as guidelines when defining the architecture structure. In addition, I will introduce architectures that have the qualities that were sought after for this application’s architecture. These architectures are generally well-known and provide good reference for my own design.

The main part of the project is the design of a software platform for laboratory instruments. Understanding the architecture does not require any knowledge of laboratory field or laboratory instruments, as this kind of architecture design can be used as a general guideline in any kind of application.

The requirements for the architecture to be designed include modularity, decoupling and domain orientation. Modularity defines that the application is composed of individual parts. Decoupling defines that the functionality of the application is separated from infrastructure, such as user interface and data storage. In domain-oriented architecture everything must revolve around the domain and the domain model layer must be isolated from the infrastructure technologies.

An aim also is to make the architecture simple to understand. Even developers do not need to understand the whole architecture, but they can already work with only limited knowledge.

2 Software Qualities, Practices and Architectures

In modern corporate environments enterprise applications are data-centric, user-friendly, scalable, distributed, component-based and mission critical. They have to satisfy hundreds or even thousands of separate requirements. In brief, enterprise applications are highly complex systems. [20] Enterprise application has also a long lifespan, and during its lifespan, the application will have many additions and changes to its functionality.

Professional Enterprise .NET defines the goals of the enterprise development as follows: reliability, flexibility, separation of concerns, reusability and maintainability. [5, 6] The ISO/IEC 9126 Standard defines a set of qualities required from a software product. These are: functionality, reliability, usability, efficiency, maintainability and portability. [4, 19] Lockheed Martin's Quality Assurance Plan also defines two important qualities, timeliness and affordability. Timeliness means the ability that software system is made available to the customer when or before it is demanded. Affordability refers to the financial cost of developing or acquiring and using the software. [1]

From these goals and qualities the goals and qualities of a software platform to be designed can be summarized as following:

- The platform has to be reliable, meaning that the application needs to function correctly.
- The platform has to be maintainable, so making corrections and adding new features should be possible.
- The platform has to be efficient, so the application's performance should be on appropriate level.
- The platform has to be flexible and reusable, which means that changes should be easy to make and functionality should be able to be used in other parts of application or in other application.
- The platform has to be delivered on time and within the budget.

2.1 Key Factors

This chapter introduces some key factors, which should be taken into account when de-signing an enterprise application. It should be noticed that the key factors follow the same patterns that could be found in the goals and qualities that were introduced above.

2.1.1 Modularity, Replaceability and Reusability

The application should be composed of individual modules, which can either function on their own or function by using other defined modules. Modules are self-contained units of code. [5, 47] A module may be only a single class, or it may be a class library which contains multiple classes [4, 77]. A module should only contain one aspect of the desired functionality (single responsibility). Changing the functionality inside a module will not have effect on the functionality of modules that are using it. Nor will making any changes to the module require making any changes to the other modules.

Modular replaceability is achieved by defining module interfaces. An interface defines what functionality is provided by the module and it must also implement all the functionality defined in the interface. Modules are not aware of each other's concrete implementations, but modules refer to each other only with interfaces. [1] Figure 1 illustrates an unwanted situation, where modularity is not implemented correctly as the modules have references to concrete implementations.

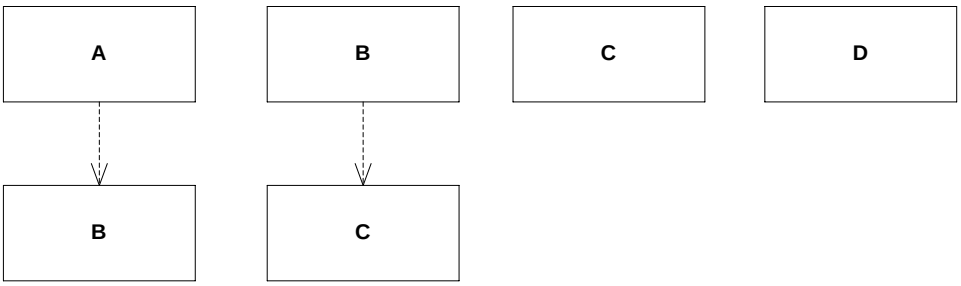


Figure 1. Modules depending upon concrete implementations

All the modules that implement the same functionality, also must implement the same interfaces. This means that the modules implementing the same interfaces are interchangeable. In figure 2. module B implements interface X and modules C and D im-

plement interface Y. Module B is using a module with interface Y, and therefore it can either use module C or module D.

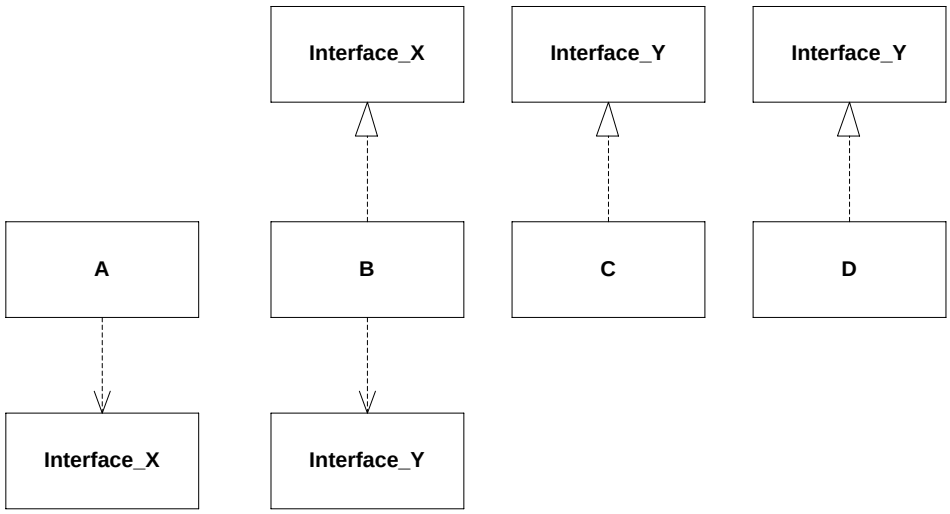


Figure 2. Modules depending upon interfaces

Replaceability plays a large role in platform design. Most of the basic functionalities do not have to know what specific implementation they are using as they all have the same interfaces. A good example of the replaceability on a class level is the Repository data source as shown in figure 3. The Repository is responsible for providing methods for getting and updating the data from the data storage. By switching IObjectContext to either DbContextAdapter or XMLDataAdapter the data can be fetched from an XML file or from a database. Classes using the Repository will not know the difference and because ObjectContext is injected into the Repository, it is not aware either, where the data comes from.

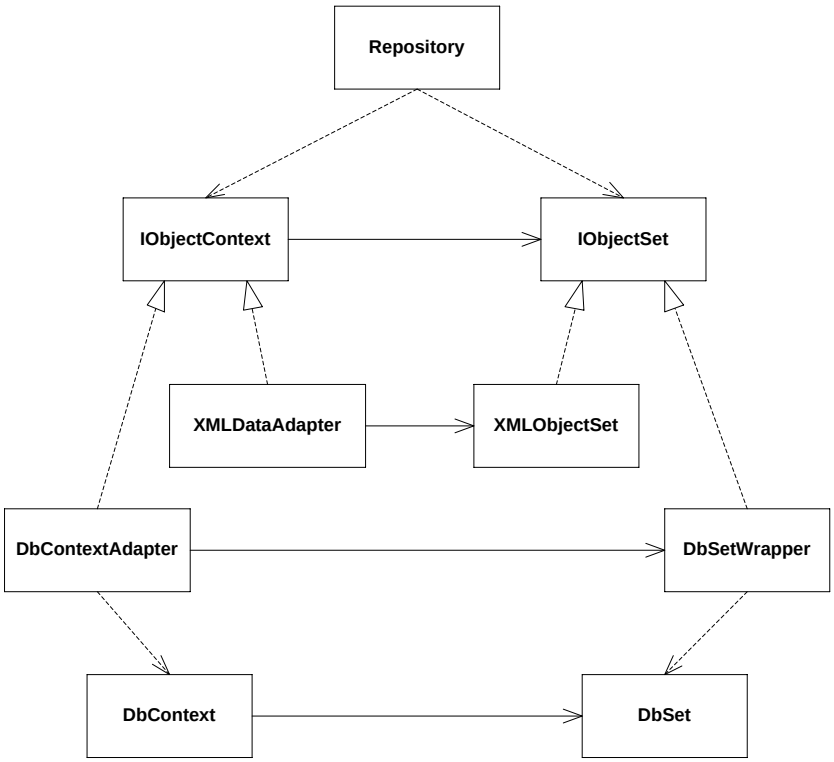


Figure 3. Abstraction of DbContext

Modular replaceability also gives an additional benefit. Often design is made in an iterative way, which means that not all requirements are known in the beginning and the design keeps evolving over time. When doing design in this way, it may occur that some of the selected designs will not work with new requirements. When modules can be replaced, there is a big chance that just some part of the application must be changed, rather than doing a larger re-write.

Another benefit that modular replaceability brings is an ability to update the application piece by piece. In the future, when new technologies or more efficient methods for executing tasks are found, the application can be updated in smaller pieces, rather than updating the whole application at once, which is usually a bad idea.

In general terms reusing means usage of base classes, common abstract classes and using the same module in a different part of the application without the need to rewrite the same code. Reusing is not to be mistaken as a copy-paste, which means the same code is just copied to multiple locations. When the same code is used in multiple locations, there is less code that can break. There is also less code to be changed when fixes are needed or new features are implemented.

Modular design also gives an opportunity to have more developers working more efficiently with the same product. Tasks can be separated more easily in their own areas, so there is not as much overlap in the application code as there would be without modularity. Developers can also work on their own modules, and they do not have to be concerned of the implementation of the modules they are using.

2.1.2 Testability and Maintainability

The importance of maintainability and testability is shown on a maintenance stage of an application life cycle. As figure 4 illustrates, on an average maintenance covers about 60% of the application's costs, as the rest covers development. With a good design, the application can have a longer lifespan, so maintenance can cover around 80%. [3, 96]

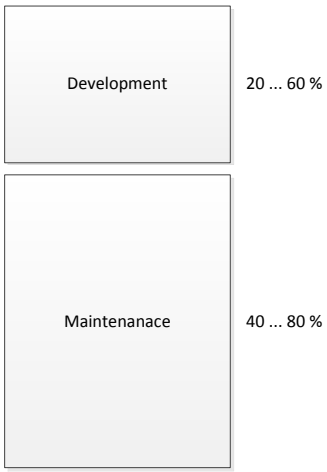


Figure 4. Development and maintenance costs

Unit testing gives confidence that the application is working as intended all the time, even after making major changes or changing the 3rd party software that the application is dependent on. Unit tests can be thought of as an insurance, because there is a cost, as more development time has to be used to get that confidence of correct functionality.

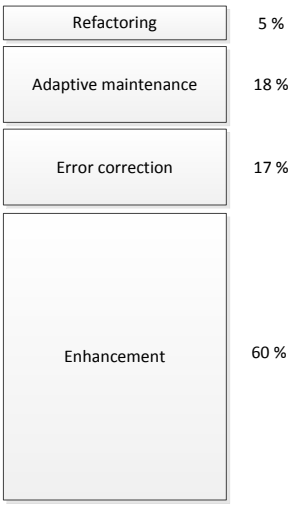


Figure 5. Maintenance cycle

The maintenance part of the application life cycle is often thought to contain only error corrections, but in reality, most of the time is used to add and improve existing features (enhancement) as figure 5 shows. Maintenance tasks, be it either error correction or enhancement, can be divided into smaller portions. These portions are show in figure 6. [3, 98]

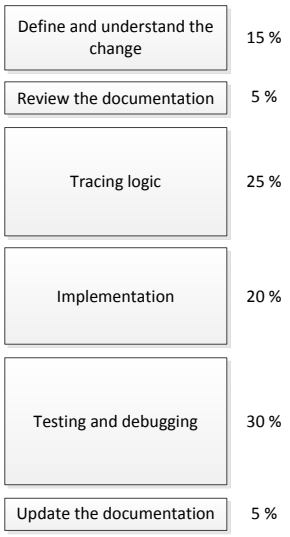


Figure 6. Maintenance tasks

Unit tests reduce the amount of time needed for tracing the logic and for testing and debugging. Unit tests are extremely useful also as learning tests. This means that the developer learns how the application works through the tests. In this way, when something needs to be fixed, the developer finds a test that covers that specific area and

debugs it through. Then he or she creates a test that will show that the bug will occur. After that, implementing a fix and validating the corrected part is much easier. The developer can also rely on the fact that the newly made fix will not break anything else, as there are other tests covering the area where the fix has effect on.

If developers are not used to write unit tests, some productivity will be lost during the initial development. As the initial development part is only a small part of the application life cycle, the lost productivity will be most probably gained back on the maintenance stage. Unfortunately there is no scientific research to prove benefits of unit testing. Usually benefit is illustrated in a similar diagram that is shown in figure 7.

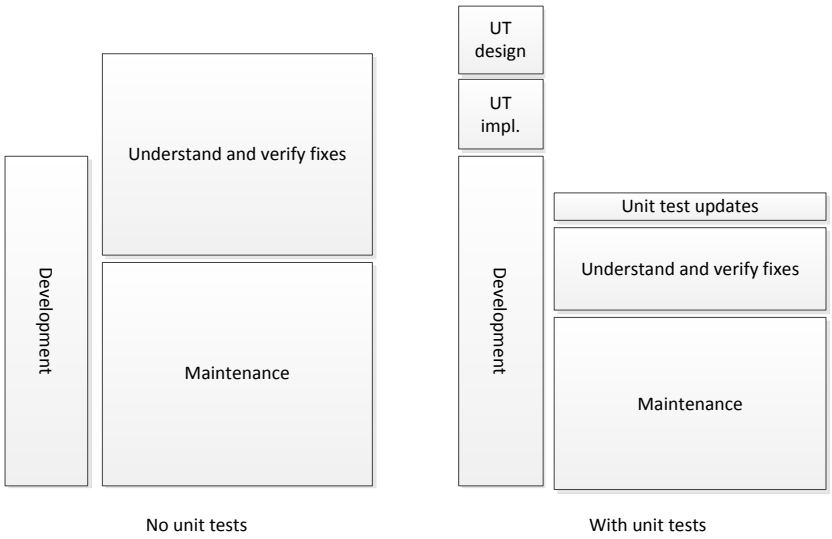


Figure 7. Time consumption with life cycle adjustment

The best way to see the benefit of the unit tests is to check how often similar bugs reappears, as ideally they should not. Unfortunately there is no way to see the benefit of the unit tests immediately.

2.1.3 Understandability

Understandability should have a big part in the design. In software production, software developers change quite often, so it is important that a new developer can start doing productive work fast. This is achieved by following the known architectures and pat-

terns and by following the KISS (Keep It Simple, Stupid) principle, which basically means that simplicity above all. [4, 131]

If a developer is not familiar with the application's architecture, he or she will familiarize himself or herself with the architecture faster when it follows the known principles. Platform design brings one big advantage. When all the products use the same platform, they all have the same architecture. When a developer is familiar with one of the products, he or she knows immediately where functionalities are implemented in the other products.

Selected technologies should be well supported and proven functional. With the technologies that are already proven functional, it is easier to find developers with a required skill set. In the programming world, new technologies come and go, so the newest unproven technologies should be avoided in enterprise applications, as they might disappear as fast as they came.

2.1.4 Parallelism

Processor clock speed improvement stopped around 2004 as can be seen in figure 8. Reason was mainly to physical issues, e.g. the processor produced too much heat or the processor has too high power consumption.

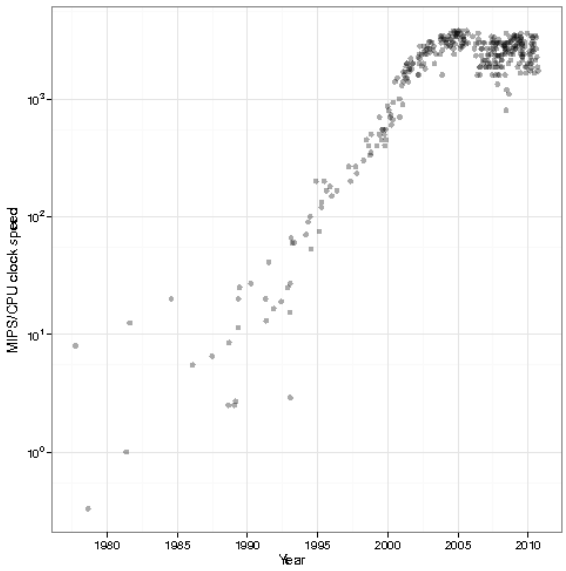


Figure 8. Clock speed improvement over time [2]

After 2004, multiple processors and multiple cores (these can be either physical or virtual) have become more common. This has been keeping the continuation of the performance increase still on the same level as just the increase in the processor clock speed. This can be seen in figure 9.

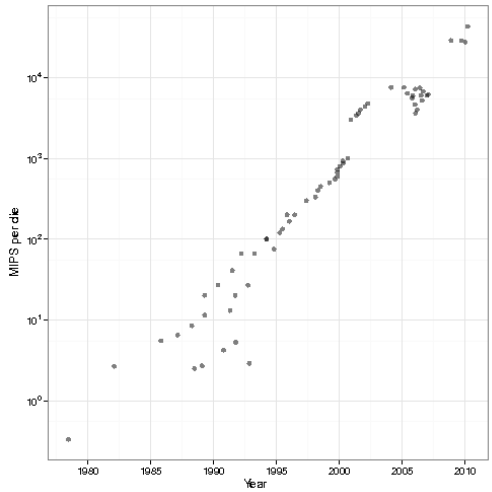


Figure 9. Effect of cores on the performance [2]

To be able to use all available performance, software needs to be able to support all available cores. This means that the program must be able to execute code in parallel. Amdahl's law, illustrated in figure 10, is often used in parallel computing to predict the theoretical maximum performance gain using multiple processors. This shows that the number of cores does not automatically bring any gain to performance. [26]

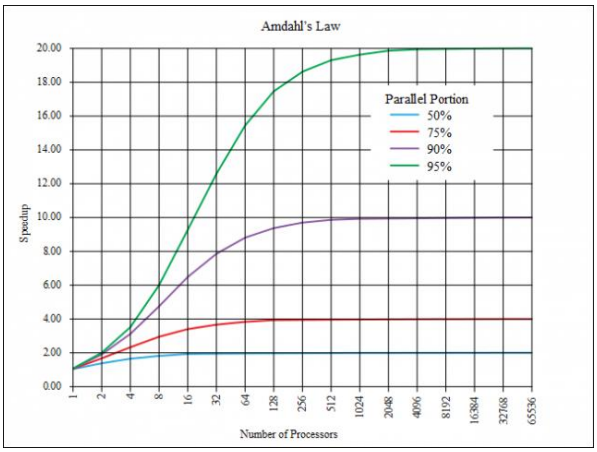


Figure 10. Amdahl's law [27]

Technologies are also becoming ready to support parallel applications out of the box. Using Microsoft .NET Framework as an example, it can be seen how threading and executing asynchronous operations are made much easier for developers. Since .NET Framework 4.0, developers have been able to use Task Parallel Library to easily execute functionality in parallel. .NET Framework 4.5 brought a more simplified approach to asynchronous programming with Async / Await. Windows Presentation Foundation's controls also got support for cross-threading, which means that controls can be updated from the background thread. The default upper limit of the threads in .NET Thread pool, as summarized in table 1, has also increased with every new .NET Framework version. Figures may vary according to hardware and the operating system. [11, 805]

Table 1. Default upper limit of threads in .NET thread pool.

Version	Default upper limit
2.0	25 per core
3.5	250 per core
4.0 32-bit	1023
4.0 64-bit	32 768

Concurrent execution must already be thought of when designing the application as adding concurrency later is often extremely hard and very error prone. The easiest way to implement concurrency is to use locks, so only one thread at a time can use specific data. A more efficient way is to have no shared data. Each thread has a private copy of all the data it needs to perform the operation. Threads should also be working on independent areas. In this way there is no need for synchronization, so the application is "wait-free". [12, 18]

2.2 Principles

2.2.1 SOLID Principles

When working with software where dependency management is handled badly the code can become rigid, fragile and difficult to reuse. A rigid code is a code which is difficult to modify. This includes changing the existing functionality or adding new features. A fragile code is a code that is a likely source for new bugs, particularly those

that appear when another area of the code is changed. If one follows the SOLID principles, one can produce a code that is more flexible and robust, and that has a higher possibility for reuse. [28, 2]

The **SOLID** principles are the following:

Single Responsibility Principle

- An object should have only a single responsibility. When a class does only one specified task, it is easier to understand and easier to modify.
- An object should have only one reason to change. For example if data storage is changed, classes that get data from data storage should not change. [29, 155]

Open / Closed Principle

- Entities (modules, classes, functions) should be open for extensions but closed for modifications. This usually means that objects can be inherited and the derived classes provide more functionality. [29, 164]

Liskov Substitution Principle

- Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.
- For example Class B inherits class A. Who uses class A, can use class B without breaking the program. [29, 180]

Interface Segregation Principle

- The dependency of one class on another should depend on the smallest possible interface.
- Many client-specific interfaces are better than one general-purpose interface. Classes should always tell as little as possible about themselves. [29, 214]

Dependency Inversion Principle

- Classes should be depending upon abstraction (interfaces), not upon concrete classes. This way the concrete implementation is changeable. [29, 201]

2.2.2 Other Principles

Low Coupling refers to the relationship of a class with another class. If classes have a high coupling, it means that changes to a class will result in changes to the other class. Low coupling means loosely coupled classes, so a class is independent of the other class. This can be achieved by using defined interfaces, so classes have no straight relationships between each other. **High Cohesion** means how well defined role and tasks a class has. With high cohesion a class only does a specified task. Low cohesion means that a class does multiple functionalities that are not related to each other. [4, 79]

In the **Command-Query Separation Principle** data storage communication is divided into commands and queries. Commands perform an action and queries answer a question. Commands change the object states but do not return values. Queries answer a question, which means returning values without changing the object state. [1]

The **Don't Repeat Yourself (DRY)** principle reduces the duplication of any information needed by the application and stores the same information only in one place. It reduces the number of times one writes the code that accomplishes a given operation within an application. The **You Are Not Gonna Need It (YAGNI)** principle means that one should add any functionality to the application only when it is absolutely necessary and unavoidable. [4, 131]

2.3 Software Architectures

2.3.1 Hexagonal Architecture

The Hexagonal architecture (aka ports and adapters architecture) is Alistair Cockburn's architecture whose main aim is to decouple the core application from the services and the infrastructure it uses. Figure 11 shows the main application, ports (hexagonal sides of the application) and adapters that are connected to the ports. [9]

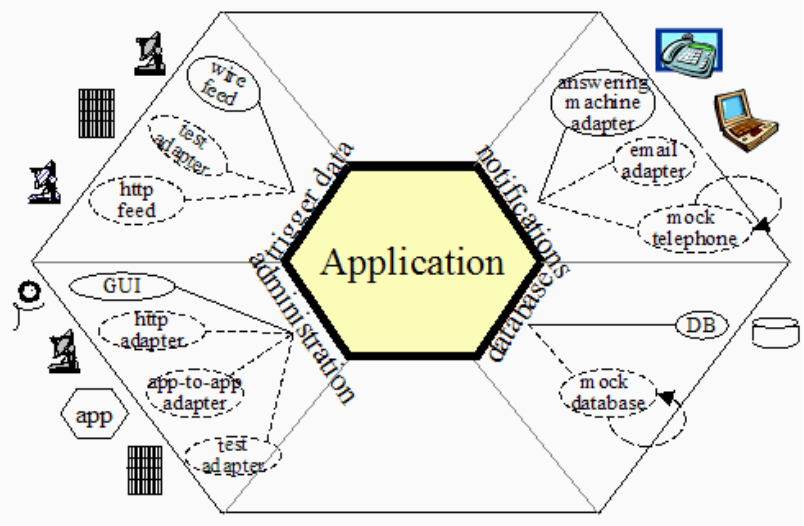


Figure 11. Hexagonal architecture [9]

Each port can be connected with multiple adapters. This allows an application to be equally driven by users, programs, automated tests or batch scripts, and to be developed and tested in isolation from its run-time devices and databases. [9]

2.3.2 Clean Architecture

Clean architecture is Robert C. Martin's articles about the need to clean up software architectures. Martin's main point is that the code should also be decoupled from the infrastructure, such as user interface, database and other frameworks. In this way the architecture is not enslaved by any software vendor. Decoupling gives also other benefits. When the infrastructure is decoupled, an application can easily be tested without a user interface or a database. These can also be replaced without affecting any other modules. [8]

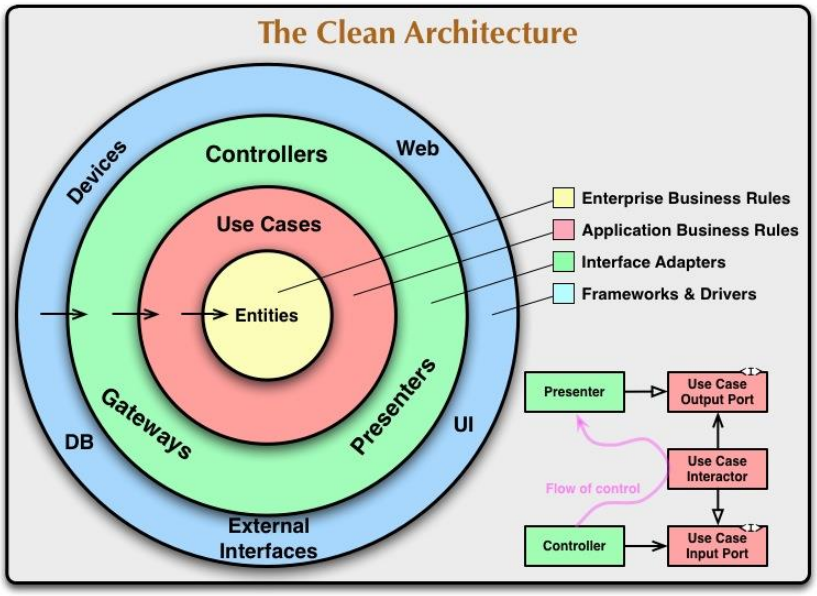


Figure 12. Clean architecture diagram [14]

Having a good architecture also gives an ability to be able to defer decisions. One should always try to have as much time as possible to make the final decision. For example, with a good design, the decision about whether to use a database or a flat file system as a data storage should not be made during design. [10]

The architecture should tell the developers about the system. They should be able to see the intention of the application, just by looking at the design and not at what frameworks have been used in the system. New programmers should be able to learn about use cases of the system, and still not know how the system is delivered. [15]

2.3.3 Domain-Oriented Architecture

In the domain-oriented architecture everything must revolve around the domain and the domain model layer must be isolated from the infrastructure technologies. Traditionally the domain-oriented architecture is presented in layers as seen in figure 13.

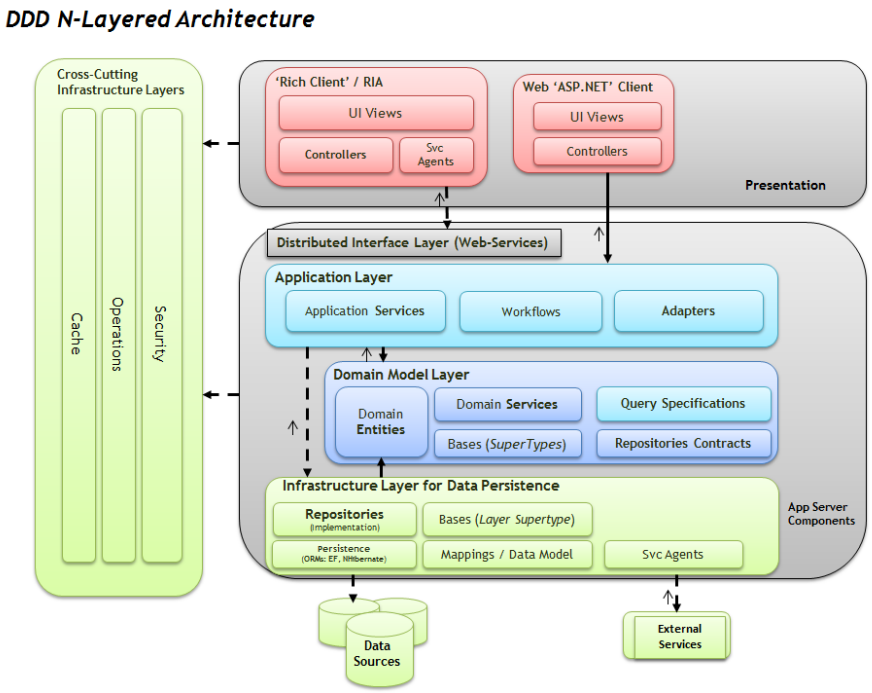


Figure 13. Domain-oriented architecture with layers [6]

It is also common to represent the domain-oriented architecture with an onion instead of layers as seen in figure 14. This kind of presentation shows in a more defined way that the domain is in the center of the application and the connections to external components are only from outer layers.

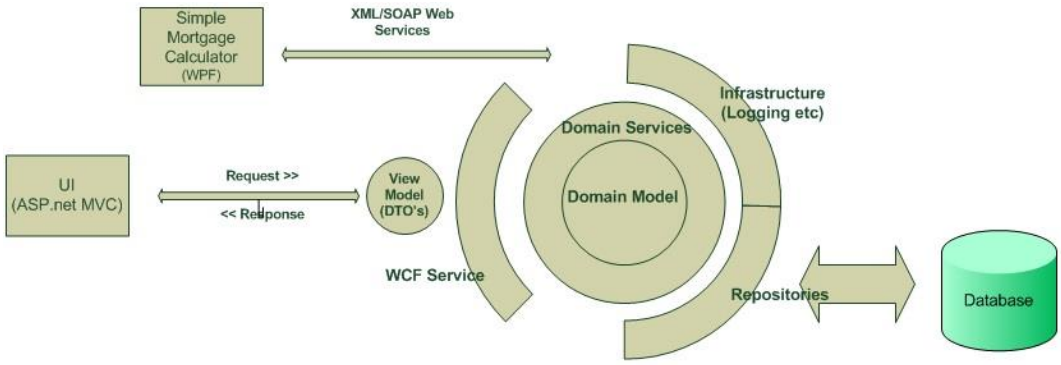


Figure 14. Domain-oriented architecture represented with an onion [30]

The domain-oriented architecture relies heavily on the dependency injection. In this way, for example, domain services do not have a reference to the repositories but only to the repository interfaces. These interfaces are often stored in the domain as connected layers have reference to it.

2.3.4 Onion Architecture

The onion architecture, illustrated in figure 15, can be seen as a combination of the hexagonal architecture and the domain-oriented architecture. In onion architecture there is a core that has the domain and application and domain services. [13]

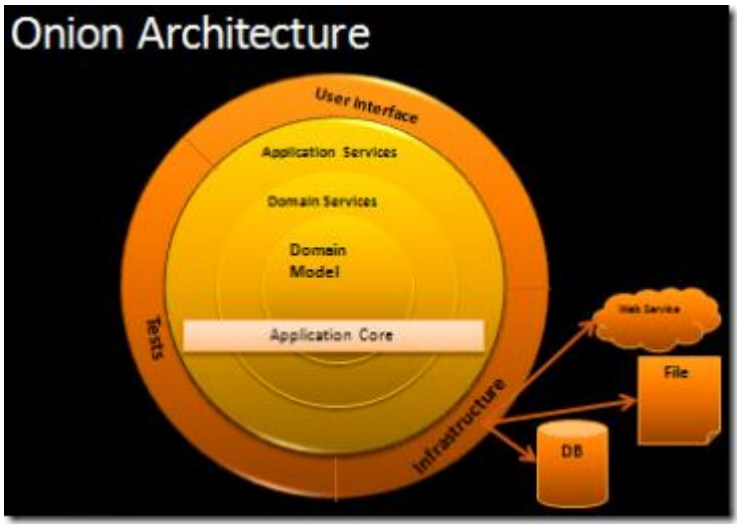


Figure 15. Onion architecture [13]

The key concepts of onion architecture include:

- The application is built around the object model.
- Inner layers define the interfaces and outer layer implement them.
- Coupling is directed toward the center.
- The application core can be run separately from infrastructure. [13]

The onion architecture is about making the domain / business logic independent of the 'inferior' factors such as data-access, user interface or services. The onion architecture does not really define how the domain is developed, but it is adamant about protecting it from outside dependencies.

2.3.5 Drawbacks

The most common drawbacks of multi-tier architectures described above are:

- Complex designs
- Reduced performance
- Complex deployment [22]

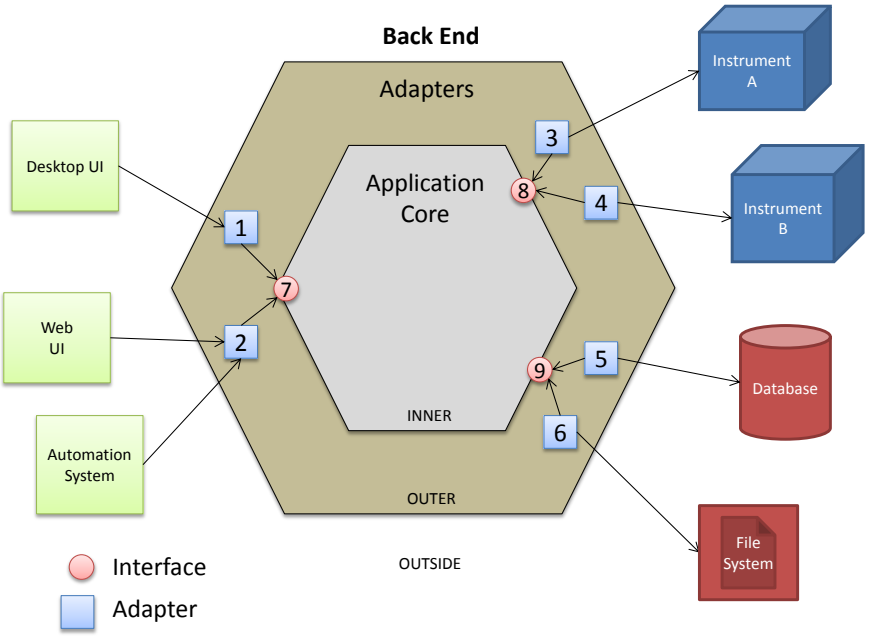
Multi-tier architectures have generally more complex designs, compared to monolithic architectures. It is always good to keep in mind that architecture should be as simple as possible. If it is known already in the beginning that the application will not need to be changed after initial development, then there will be no need to make complex designs.

Complexity usually also means reduced performance. Implementation that is designed to do only one specific task is most often also fastest, but of course making this kind of implementation design is not always the best option. There has to be a balance between performance and having a design that is able to have changes that are possibly needed in the future. Physically separated tiers may also have a great impact on the performance and this is something that does not always come up during development time.

Deployment of the application should always be as easy as possible. However if the application is distributed to multiple locations, it will be almost impossible to have a simple deployment. It is more likely that each separate location will require its own installation and defining configurations as to how to connect to other separate locations.

3 Overall Platform Architecture

Designed platform has a combination of the hexagonal architecture and the domain-oriented architecture. Figure 16 illustrates the back-end which contains application core, adapters and ports, and the infrastructure (outside part in figure 16) which contains user interfaces, data storage and laboratory instruments.



Adapters	Interfaces
1. Façade 2. Web services 3. Instrument A 4. Instrument B 5. DatabseContext 6. FileSystemContext	7. Application service interfaces (each service has an own interface) 8. IInstrument 9. IDataContext

Figure 16. Hexagonal architecture

Platform's architecture aims to make the application core independent of low-level functionalities, such as data storages, user interfaces, object-relation-mappings and 3rd party components. The benefits are that the architecture and the code become testable, changeable and understandable.

Architecture relies heavily on the dependency injection principle. It means that high level modules should not depend upon low-level modules and both should depend upon abstractions (interfaces). These abstractions should never depend upon details and

details should depend upon abstractions. The back-end architecture is domain-oriented and the key principle of the domain-oriented architecture is that the domain model depends on nothing and everything depends on the domain model.

The architecture can be divided into tiers which can be seen in figure 16:

- Front end (green)
- Back end
- Data storage (red)
- Instrument (blue)

From the tiers, only the instrument is always physically separated. In the most common case, when the application is a standalone installation, front-end, back-end and data storage tiers are on the same computer. It is also possible to use centralized data storage and then the data storage tier is also physically separated. When the back-end acts as a centralized service, then the front-end and the back-end are physically separated and the data storage tier can also be physically separated, depending on whether centralized data storage is in use. Instead of using the concept about tiers, this architecture uses separation into back-end and front-end, as seen in figure 17. In general terms, the front-end is what the user can see and the back-end is what the user cannot see.

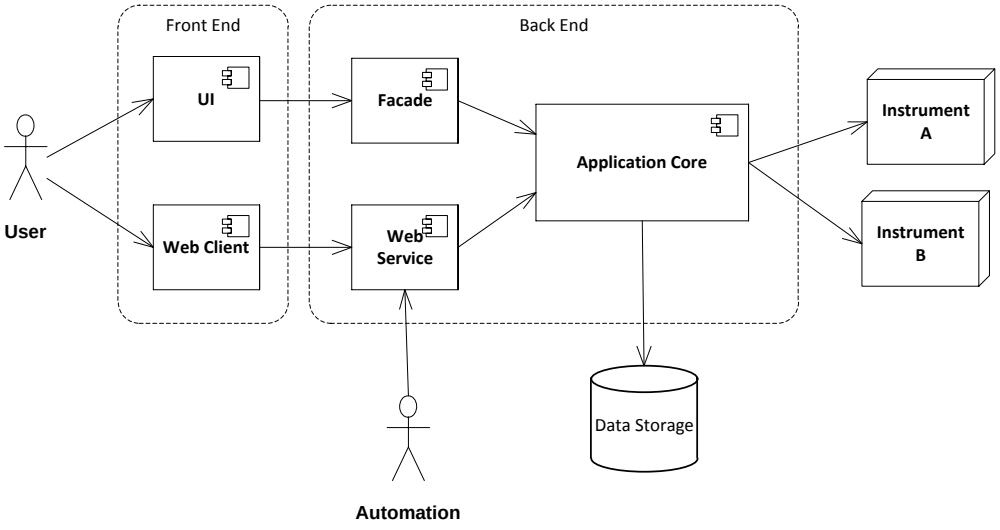


Figure 17. Back-end and front-end areas

Application functionality is in the back-end, so it is possible to execute the application logic without the user interface. This kind of design helps to separate user interface and

create more modular design. The front end's (user interface) main purpose is to show data, help a user to create commands and execute the selected functionality. This kind of front end that does not have a data saving or heavy data processing functionality is called a hybrid client.

The front end is only aware of the façade module from the back end. The rest of the back end functionality is a black box for the user interface. The façade module has only asynchronous methods. In this way it is impossible to use blocking methods and make the user interface not responsive while executing the back end functionality. Web clients and automation front ends use only the web service module. These methods are synchronous, as it is a common practice to handle synchronization in JavaScript.

A software platform is a set of subsystems, which form a common infrastructure. The platform provides a common functionality from user authentication to data storage access. Related software products are developed on top of that infrastructure. The platform defines initial architecture structure and how related software products will be structured.

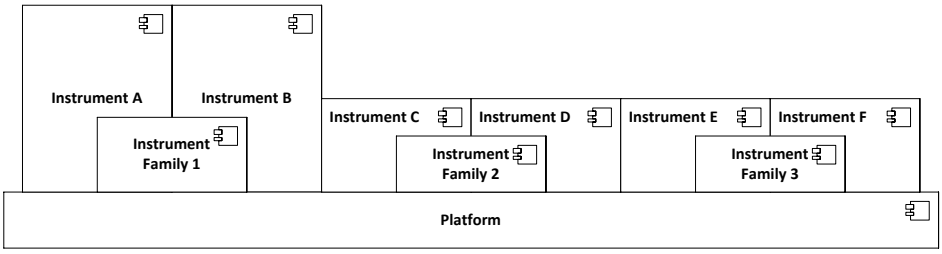


Figure 18. Platform and more specific modules

Platform modules are extended with instrument family and instrument specific modules as seen in figure 18. Functionality is kept on a lowest possible level, so higher-level modules have the possibility to use the same functionality. When the same functionality is shared, adding new features and fixing old bugs will be done to only one location.

3.1 Back-End Architecture

Back-end has domain-oriented architecture. In the domain-oriented architecture everything must revolve around the domain and the domain model layer must be isolated from infrastructure technologies.

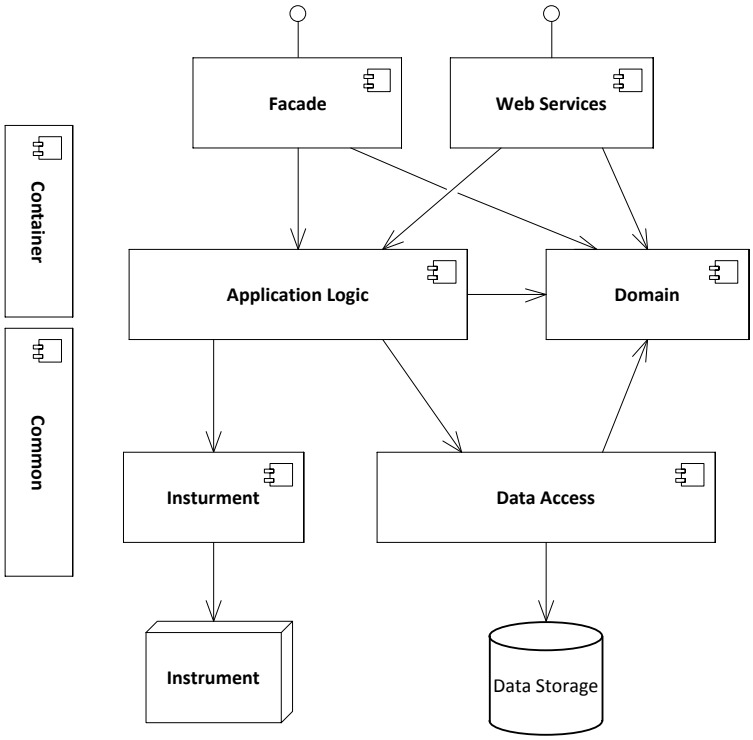


Figure 19. Back-end layers

The back-end architecture can be divided into layers. The application logic layer contains services, calculations and execution logic. The domain itself is considered a layer. The infrastructure layer provides communication to external components. Data access module is communication to data storage, which is often also called the data access layer. The infrastructure layer also contains façade, web services and instrument. The façade and web services are communications to user interfaces and the instrument is communication to the physical laboratory instrument. The container is also considered a part of infrastructure as it has 3rd-party dependency injection container.

3.1.1 Back-End Modularity

Instead of layers, it is more natural to divide back end into modules. Back-end modules have straight reference only to the domain module. Reference to other modules is upon interfaces and a reference to concrete implementation is through dependency injection.

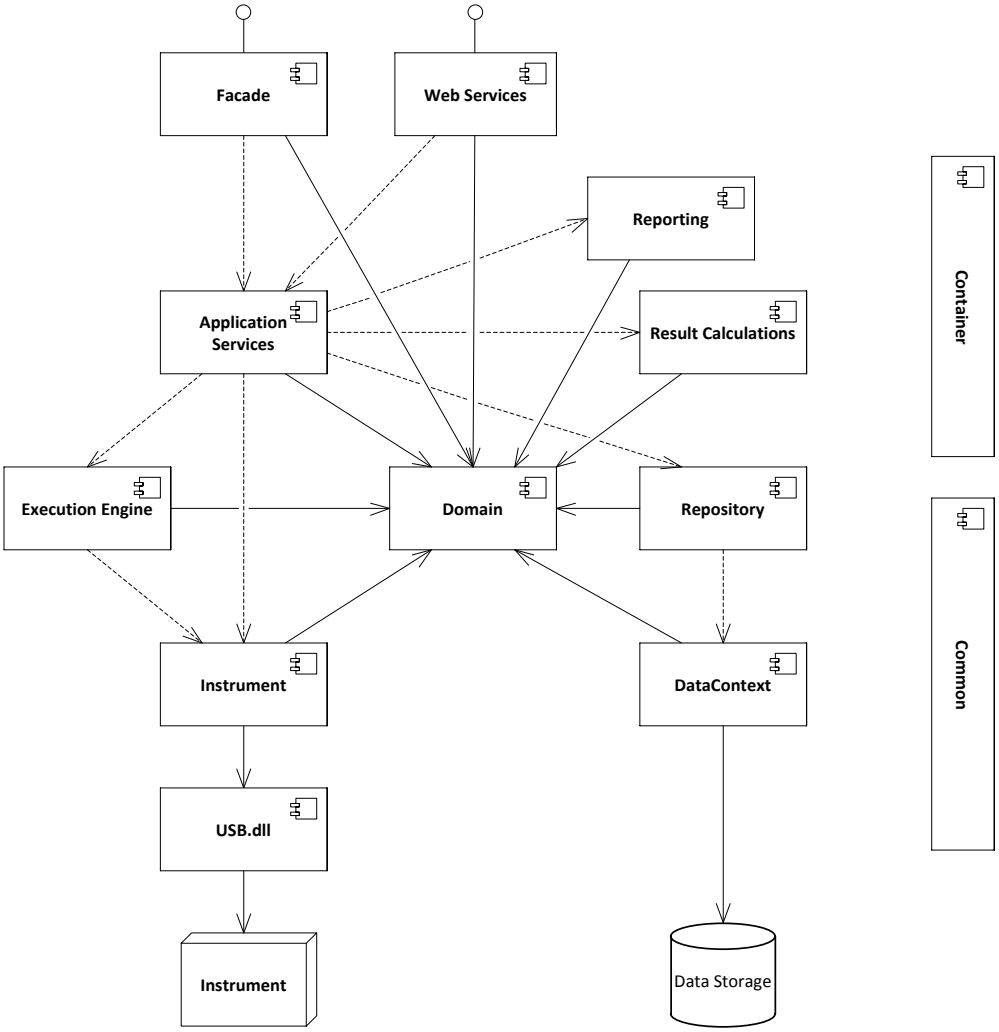


Figure 20. Back-end layers expanded to module level

Figure 20 shows references between modules. A straight line is a reference to a module and a dotted line represents dependency upon interface.

The Domain has data models, which are called domain models. Domain model classes represent real world objects. Most of the domain models also map straight to database tables. The domain has also interfaces for classes in other modules and it has functionality that only needs other domain models. This kind of functionality is for ex-

ample validation, rule creation and creation of new object. The Domain does not have references to any other modules.

Application Services can be thought of as an entry point to the application and it manages its own concrete part of the application. Services communicate with other modules in the application. Services handle user authorization, fetching data from data storage, validating updates and processing other user requests. The module has a functionality to save data asynchronously to data storage. External components (user interface, web applications and web services) can use services through the façade and web services.

The Execution Engine handles protocol execution, asynchronous command execution to the instrument and asynchronous response handling. **The Instrument** is an object model of the real laboratory instrument. It contains a mapping functionality for mapping domain commands to real instrument-level commands. The module has a functionality to find instruments that are attached to the computer.

The Reporting module has a functionality to create reports and different kind of data exports. The most commonly used report format is the PDF, and the export formats are to either Microsoft Excel or in raw format to a text file, so users can parse data as they want.

The Result Calculations module has a calculation and result handling functionality. Calculations are implemented in plug-in style, so in the application initialization, plug-ins are loaded dynamically. Because of the plug-in design, this module also has domain models for results steps, which are used to save the wanted calculation steps to data storage.

The Data Context is a connection and a model of the data storage. It has the data storage manipulation functionality. The module has configurations to configure mappings of domain models to data storage. The Data Context also implements the unit of work, which means that it keeps track of the changes and saves changes as a batch to data storage. **The Repository** provides a search and saving functionality to data storage. It is a middle layer that abstracts data context from the application.

The Façade module's main task is to handle all operations from the desktop user interface to the back end. It is also responsible for creating asynchronous operations, deciding what to do when data is ready (e.g. what to do if the view is not visible any more) and pre-fetching data based on previous actions. The Façade module has a factory, which is used to create new facades. **Web Services** is a handler for all operations to the back end from the web user interface. It executes functions directly from the application services.

The Container module has a dependency injection container and mapping information between classes and interfaces. The module also contains a mechanism to dynamically load all assemblies that are instrument-specific containers. The Container module has shared instance of the container that has platform instances. It is shared by the whole application. The module also has a creator for creating new containers based on instrument's product identifier.

The Common module has helper classes and functionality that can be used anywhere in the application. It does not have reference to any other module and it does not have any application specific functionality. **The USB** module handles sending and receiving data through USB interface. It doesn't have any application related logic.

3.1.2 Back-End Internal Communication

Modules do not have any generic method to communicate with each other. Classes inside modules have events which are used to notify the other classes in other modules when some changes occur.

The application service module has an event aggregator, which is used for sending messages between the same services. For example when a session is opened in one session service, it can send a message what was just done, to other session services instances. The communication can be seen in figure 21.

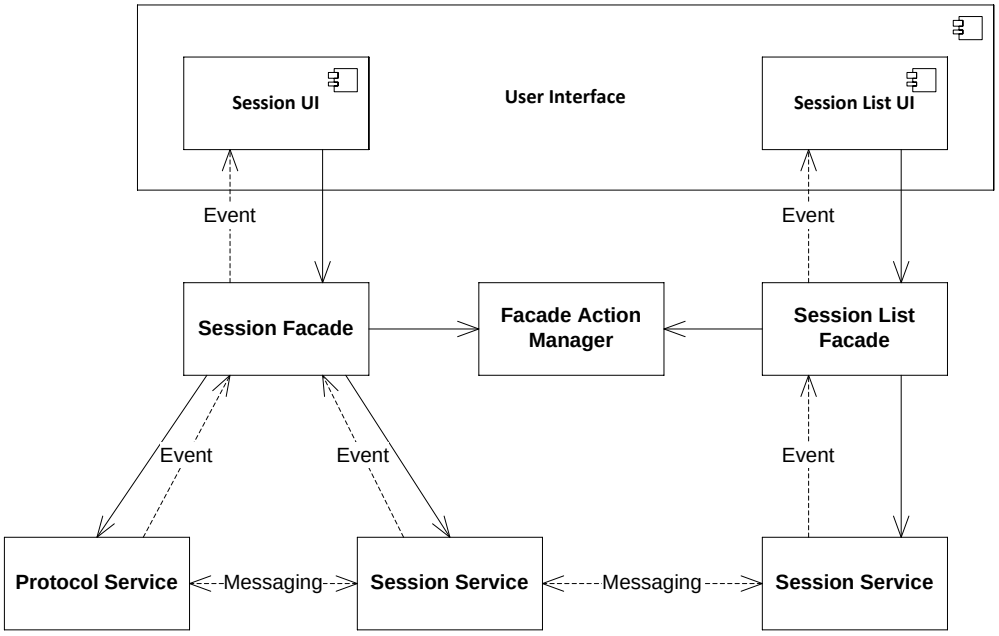


Figure 21. Internal communication

The façade has a manager class which keeps track of simultaneous actions. The manager class's main function is to prevent executing too many simultaneous actions, which would slow down the computer. The manager class does not prevent a user from starting as many actions as he or she desires to, but only prevents the application from executing too many preloading or pre-fetching tasks.

3.2 Front-End Architecture

A desktop client is developed with Windows Presentation Foundation using the Model-View-ViewModel pattern. When using the MVVM pattern, the functionality should be in the ViewModel, so adding the functionality to the code-behind must be well reasoned. The main concern is to keep the user interface responsive all the time. Performing an operation can take a long time, but during these operations the progress dialog is shown to the user.

3.2.1 Modularity of the User Interface

The user interface is divided into modules. Each module contains Views, ViewModels and all the classes the module requires. Modules are designed in a way that they can function on their own as small applications if needed. The main application passes instances of a façade or a façade factory and an instance of an event aggregator to the module when the module is initialized. The module composition is illustrated in figure 22. By default, modules have views for all instruments and the content is loaded dynamically based on the technologies and features of the connected instrument.

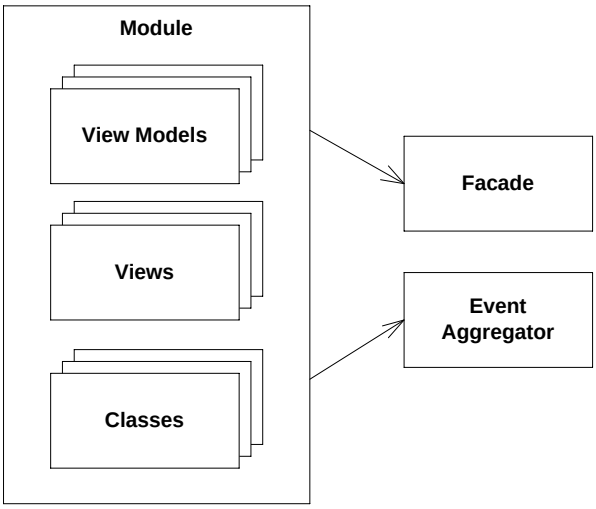


Figure 22. User Interface module

The Main module has a main application where all other parts of the application are loaded from independent modules. The module also has a bootstrapper, which has the application initialization functionality. It initializes the service locator for the user interface and dynamically loads all configured modules.

The Common module has shared user controls for windows and ribbons. The common module has also base classes for ViewModels and generic helper classes. The module does not have reference to any other module of the application. **The Localization module** has only translation information for controls. Localization data is stored in resource-files, and the controls refer to correct data by its identifier.

Each module has a specific functionality. **The Session module** has views and functionality for creating and modifying measurement parameters and configurations and

for handling and reviewing measured results. Session is a dataset, which encapsulates all this information to a single set of data. This is by far the largest user interface module as it includes most of the functionality that the user uses by the application. All views and controls are in one module, because they are all needed for the application to perform as intended. **The Session browser module** has a view for listing and searching for sessions. It also has a functionality to store sessions into folder structure and to organize this structure. **The Recent module** has the functionality to list recently opened sessions. **The Instrument module** has an instrument list view and instrument information views. **The Settings module** has a settings view. Most of the settings are stored in the Back End, but some user interface related settings are handled in this module.

3.2.2 General Design and Regions

The main navigation is handled with tabs. When the application is loaded there is one home tab. This view has a list of recent session, functionality to create new sessions and basic information of the application. When a session is selected or a new one is created, a new tab is opened for that session. Also settings and instrument lists will be opened to a new tab.

Prism (composite application library made by Microsoft) is used to provide region support for the user interface. Regions are used for placing content dynamically to correct locations. In this way, modules themselves do not define where views are places, but the main application decides what to do with the view it receives from the module.

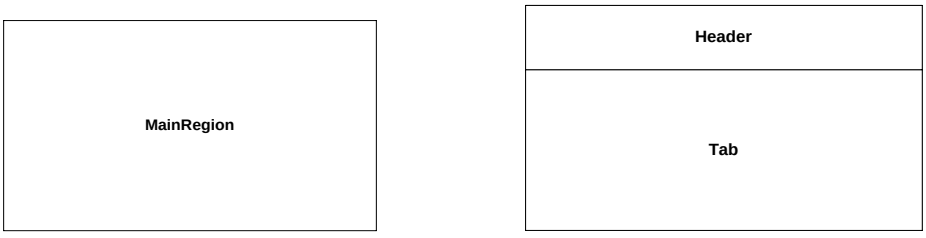


Figure 23. Shell view and main view

Shell is the term for the main window of the application. As shown in figure 23, it has only one region and no content at all. The main view has regions for a header and tab.

The header will include the main toolbar and the ribbon and the tab region will have tab control. Separation of the shell and the main view is used to decouple the main window initialization functionality from the actual application views. Normally a module contains a main view, which will be placed to the tab region, and the module may also contain a view that is placed to the ribbon. Figure 24 shows the session modules views.

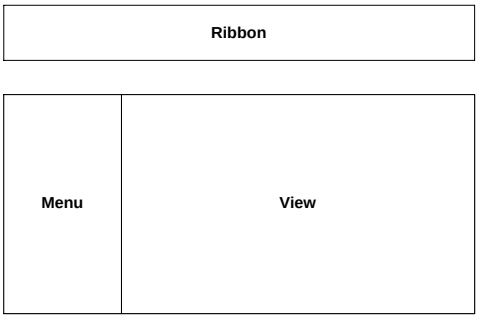


Figure 24. Session modules views

The module is responsible for sending the correct view when the main application requests a view from the module. A module may contain a view that has any number of child views. For example the Session view has a menu that contains all the available views, and the selected view will be shown in the view region. The session module also has a ribbon view which is placed to the header region. The session browser module contains only one view that has a tree view of session and folder structure.

3.2.3 Application Responsiveness

As Windows Presentation Foundation is not known for its performance, attention must be paid to keeping the application responsive. This is done by pre-loading controls and required assemblies and keeping the application responsive in every situation.

Per-loading of controls is done before the main window is created. This is done because Windows Presentation Framework lazily loads assemblies it requires to show the controls and this load operation takes a small amount of time when done for the first time. Each module implements a method which returns a set of controls that require pre-loading.

Keeping the application responsive means that when a user does some action that requires showing new content, first the user is navigated to a new page, then a progress dialog is shown to the user and lastly new content is loaded on the background. When new content is being created, the data is being loaded on its own thread. Depending on the page and type of data it has, the progress dialog is removed either when the page is loaded or when the data is ready from the back end. In this way a user has a feeling that application is doing something all the time and it is not just freezing on longer operations.

3.2.4 Application Initialization

During initialization a splash screen is shown to a user. The background thread is responsible for loading data and when this thread is ready, the main window is created on main thread. The application initialization includes the following steps:

- Show splash screen
- Load found modules
- Initialize the back end
- Start data preload
- Preload controls
- Create main window
- Hide splash screen
- Show the main window

The data preload makes predefined queries to the database. In this way most common queries are cached and this will improve the Entity Framework's performance. The data preload is continued on the background even after the main window is shown. Preload tasks are executed only when the user is not doing any actions that require fetching data from the back end.

3.2.5 Internal Communication of User Interface

The ribbon view and the session view will communicate, for example by a button click, with delegates. A delegate is a C# type that encapsulates a method (similar to a function pointer in C++).

The main application and modules exchange data with delegates and with an event aggregator. When a new module is loaded, it publishes an event with the event aggregator. The event contains information of the module, including a delegate, which returns the module specific views. The manager class is subscribed to that specific event and then decides what to do with the received information. In this way views are not loaded before they are actually needed. Figure 25 shows activity when an action for loading a view from another module is made.

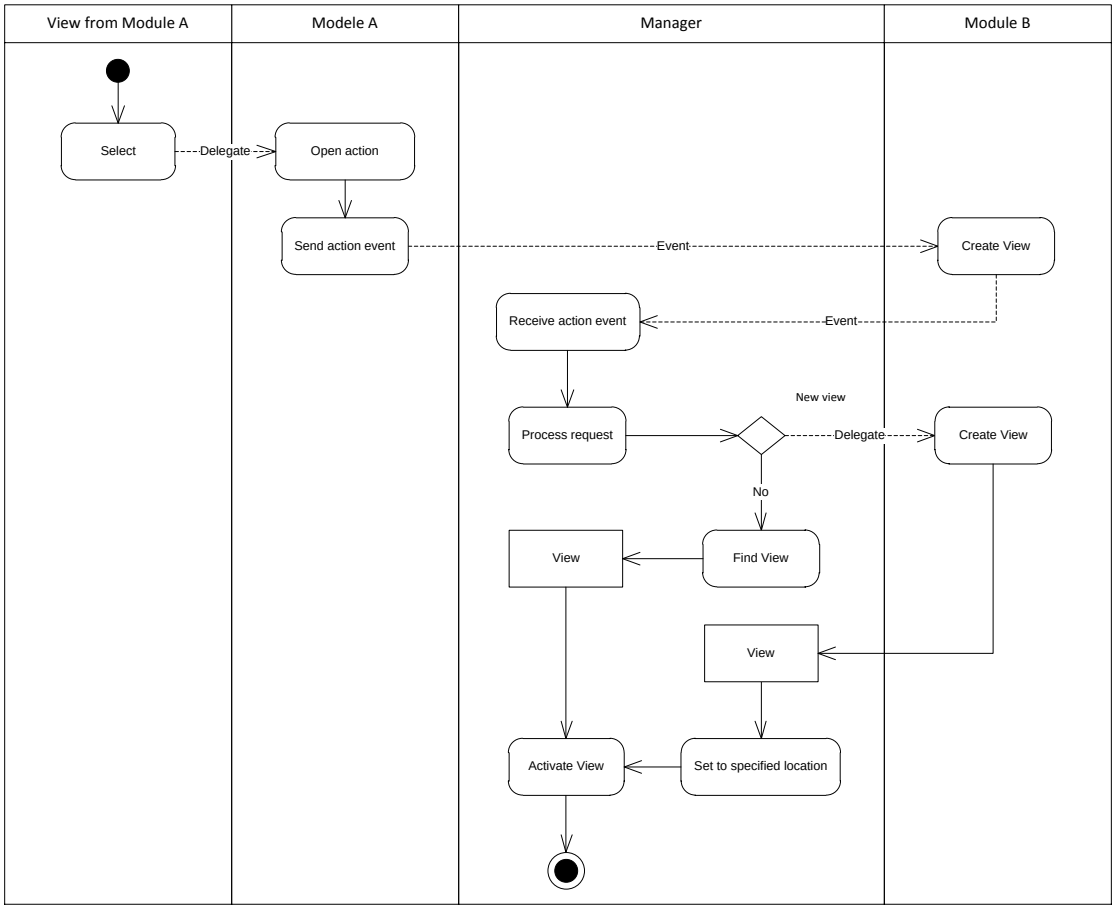


Figure 25. Module activated from ribbon

Modules use the event aggregator for exchanging information. When an action is performed in a module, it publishes a new event. Another module has subscribed to that event and will perform an action based on the data it receives. For example in figure 26, Module A notifies with the event aggregator that it would like to execute some action. It will not know if any module is subscribed to that event, but in this case, module B is subscribed to the event and will execute some functionality. In some cases module

B might send another the event with event aggregator notifying that it has executed requested function, so then Module A will know that the request has been processed.

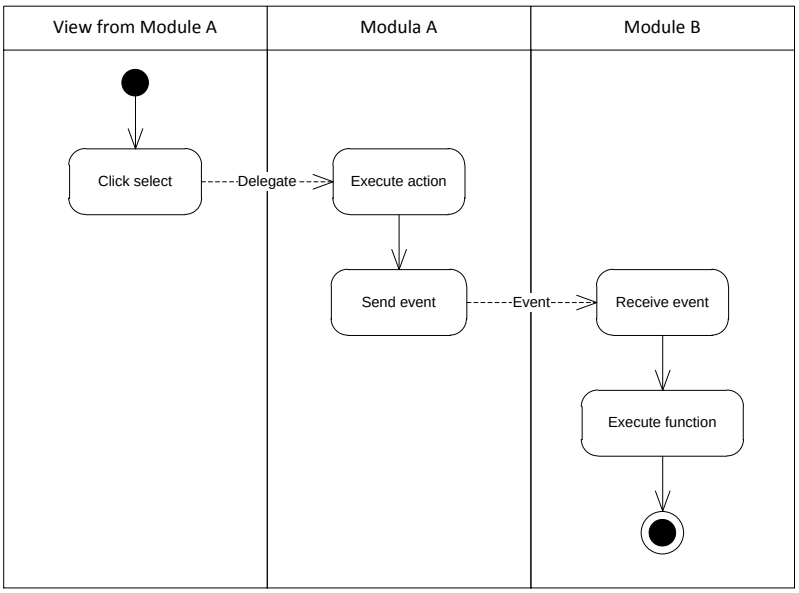


Figure 26. Two modules performing functionality

The modules do not know who is subscribed to the events, so it is the developer's responsibility to make sure that the module subscribes only to events that it really needs and will not perform any unnecessary functionality. If this rule is not followed, the application's performance might suffer and it might result in unwanted behavior.

When a module needs to notify something to the user, it will send a message with the event aggregator, rather than just showing a message box. In this way the application decides, depending on the notification type, how the user will be notified.

3.3 Platform Design

The platform level is designed to provide the basic functionality and services for the application to perform basic functionality. Platform level classes implement an interface, and instrument-specific classes inherit platform level classes. Instrument level classes can also use platform level classes if all needed functionality is already on the platform level. This is illustrated in figure 27.

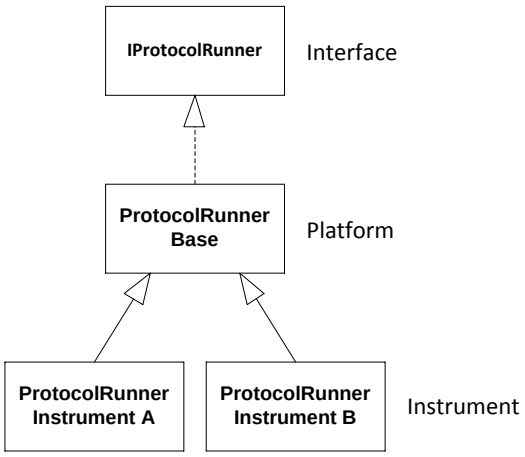


Figure 27. Common instrument-specific inheritance

Instrument-specific classes and modules may override all platform-level functionality if needed. In this way the platform defines only the architectural structure, but the instrument-specific level defines all functionality. The instrument-level can also contain classes and modules that no other instrument can use.

3.3.1 Dependency Injection Container

The dependency injection container acts as a service locator and as a dependency injector. The DI container is not passed around to other classes, but it is only used when initializing a new session. In this way, what classes are allowed to be used and allowed to do, can be handled better.

Most of the classes have a new instance passed when the instance of the class is requested from the container, but some instances are shared within the application. Shared components are loaded to the container when the container is created.

Dependency injection container initialization includes the following steps:

1. Create a new container with the correct catalog.
2. Inject shared instances to the container
    a. Instrument service
    b. Messaging
    c. Logging
3. Set serial number for the instrument.

Shared components are the instrument service which is responsible for maintaining the attached instrument list, logging and messaging.

The dependency injection container has configuration definitions which contain the information of what concrete classes are returned when an implementation is requested from the DI container. The platform level has shared configurations, and the instrument specific level has its own additional configurations. These two configurations are combined to make a full type catalog.

Configurations are loaded automatically when the application is initialized. Each container module has module information that defines to which instrument serial number the container is associated. When a new session is created for the instrument, the correct type catalog for that instrument based on the instrument's serial number will be used automatically.

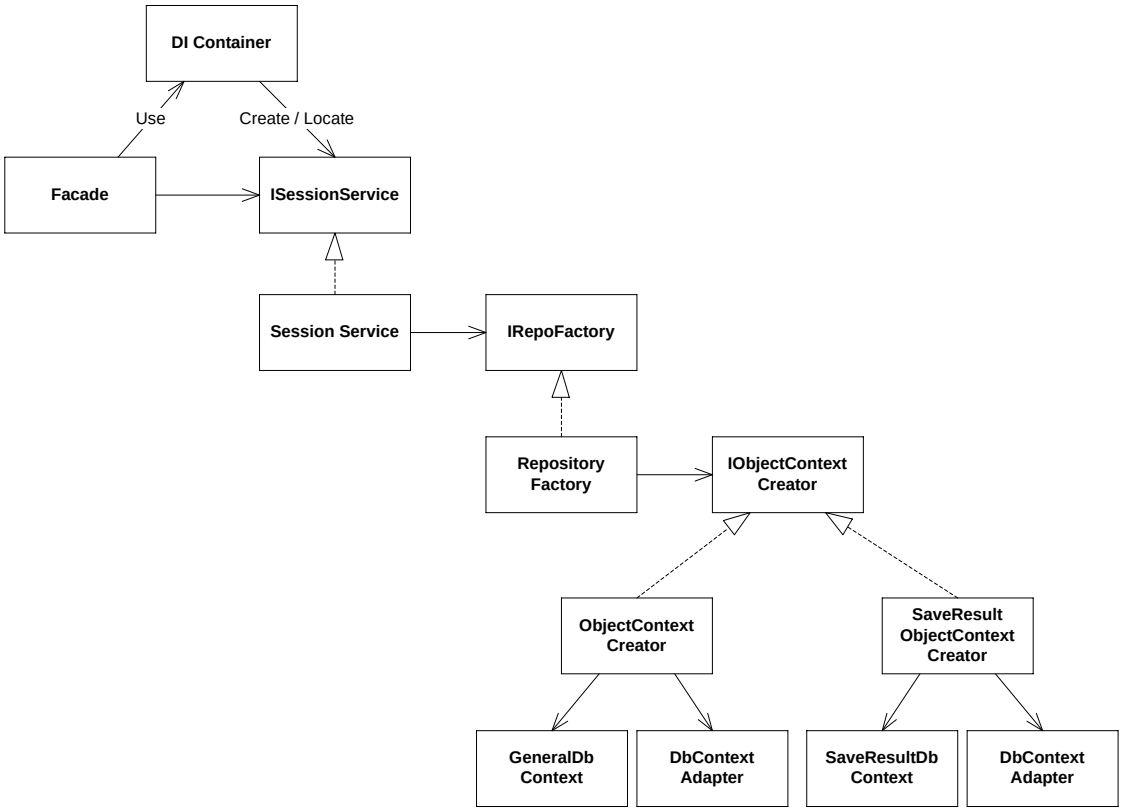


Figure 28. Façade requests ISessionService implementation from DI container

When an implementation is requested from the DI container, a set of objects is created. Figure 28 shows that when the interface ISessionService is requested from the DI container, a new session service is initialized. The session service uses interface IRepoFactory, so the DI Container will initialize a new repository factory. The repository factory uses the interface IObjectContextCreator and depending on the configuration, a selected creator will be initialized.

3.3.2 Instrument-Specific Modules

Each instrument can have *instrument-specific* services and repositories. Figure 29 shows the platform and two instruments, service and repositories they have. Instrument specific services and classes can also use other classes that are defined only for that specific instrument.

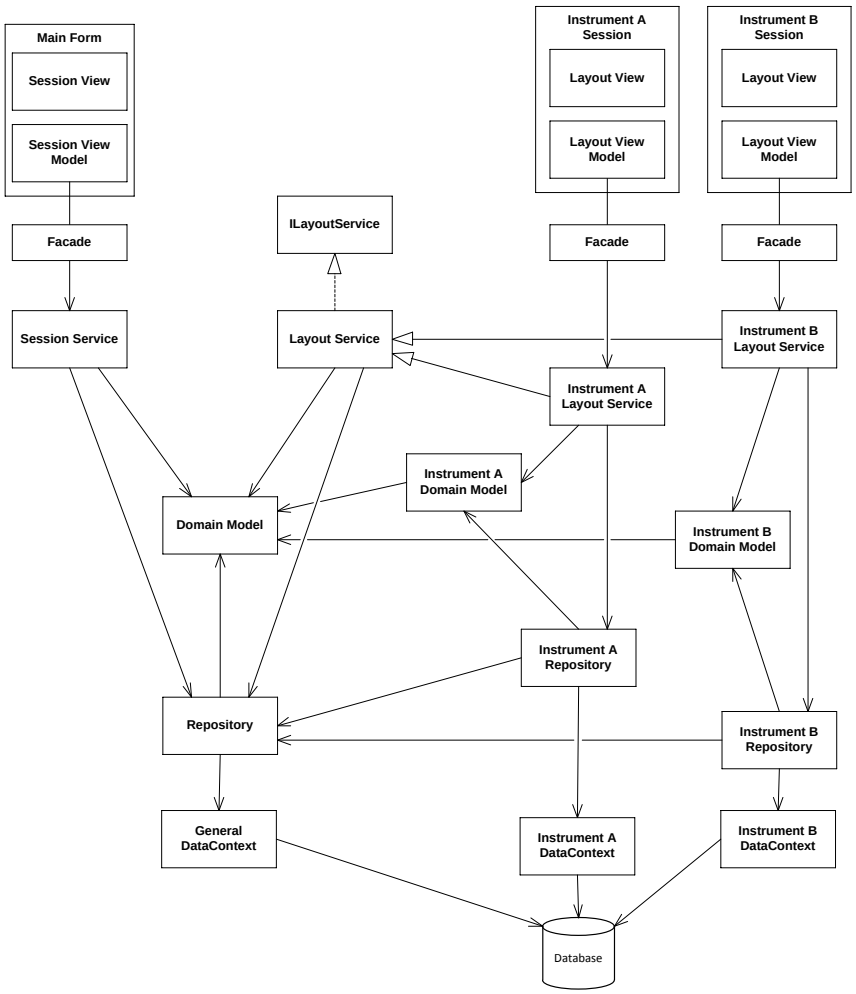


Figure 29. Instrument-specific modules

Instrument-specific domain models are only needed by data context, which gets the data from the database and by the class that uses instrument specific-data. For example the protocol repository does not need to know any instrument-specific data. It can get and update data models without knowing instrument specific model implementations.

Instruments often use only platform-level implementations because those already have all the functionality that is required. For example in measurement execution, which can be seen in figure 30, the whole service and most of the execution engine modules are from the platform.

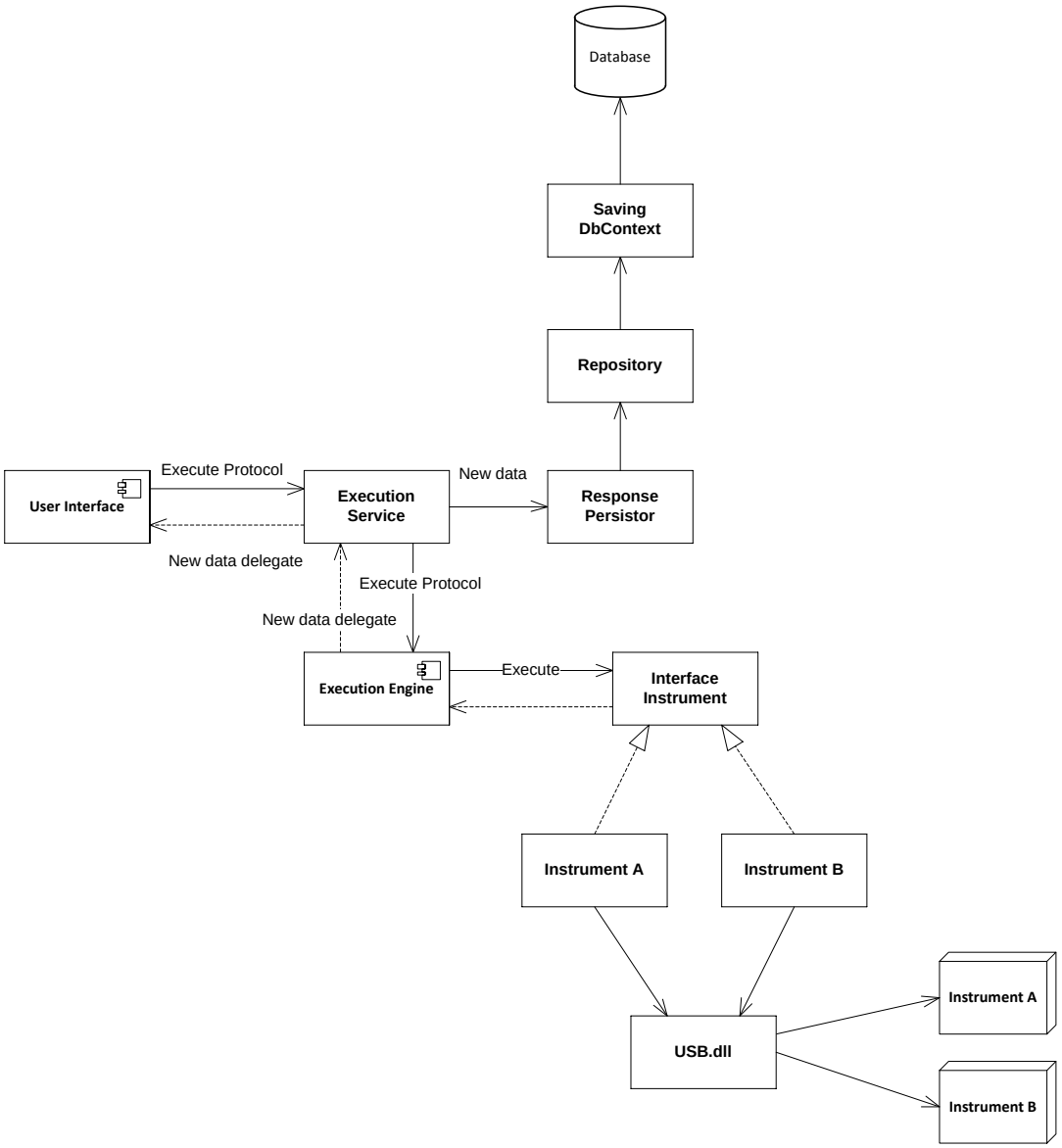


Figure 30. Measurement execution

Most of the instrument-specific functionality is related to how the specified measurement is mapped to instrument commands, and how responses are handled from the communication layer.

3.3.3 Back-End Environments

The user interface does not share one instance of the back end. Each user interface session has its own environment, which provides it with all the needed functionality. The environment is created with the dependency injection container. As mentioned in the section 3.3.1 when a new container is initialized, it is loaded with correct data. This is then used to create all needed services and classes that act as an environment. Figure 31 shows environments for the main application and two different sessions.

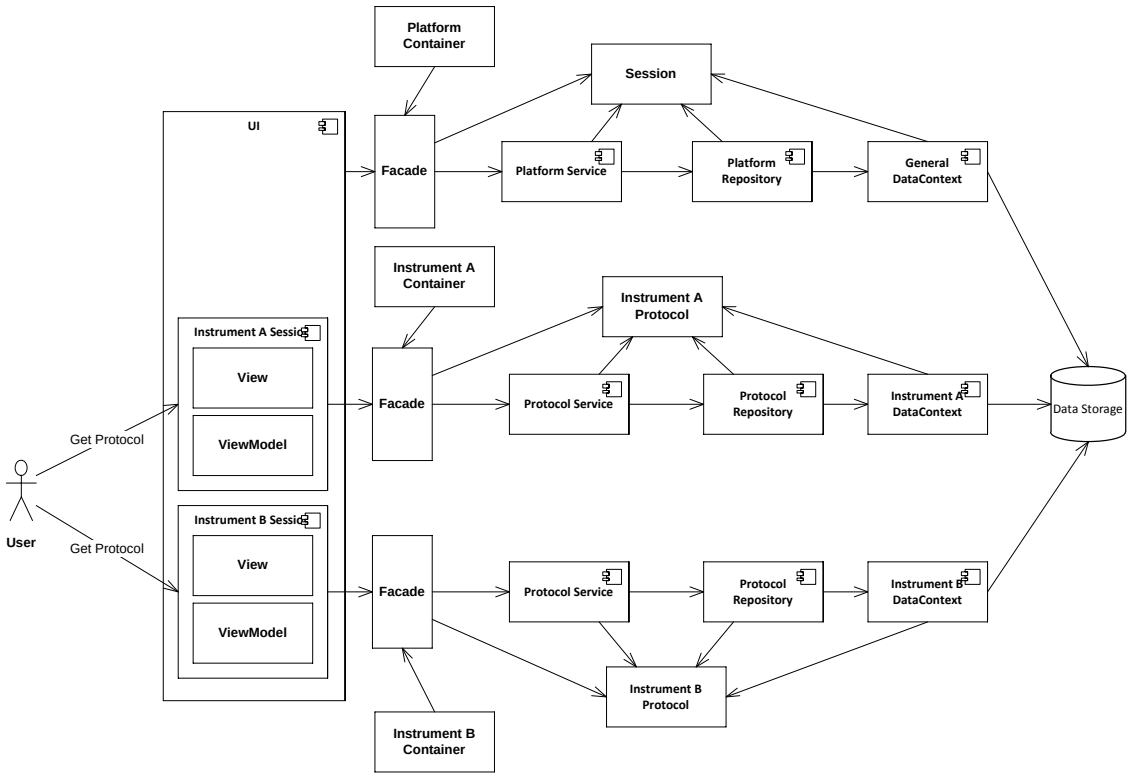


Figure 31. UI and back-end environments

Having a separate environment for each session provides a more efficient system. Services do not have to wait for other tasks to finish, because they have only one session to serve. This kind of approach requires more concurrent processing power from

the computer. This is also closer to how web applications behave, as for web applications, new instances of the classes are created for each request.

A session data can be considered immutable as an only valid state of the data is in the database. Data is fetched from the database and processed to objects. Then it is passed to the user interface and no local copy is kept in the back end. This kind of approach prevents multiple handlers from modifying the same data objects simultaneously. The back end prevents multiple environments from having editing access to the data by marking the session as locked. In this way the only one environment at a time can have edit access to the session data.

The user interface may share the same data objects between different View Models, and then it is the user interface's responsibility to be sure that modifications are notified between View Models correctly. Back-end services may notify when changes are made to some specific data, but again it is the user interface's responsibility to validate that data it still has is valid.

3.3.4 Independent Modules

Having independent modules gives the advantage of using only the selected ones, instead of using the entire back end. All modules have a reference to the domain module, so this must be always attached with the selected modules.

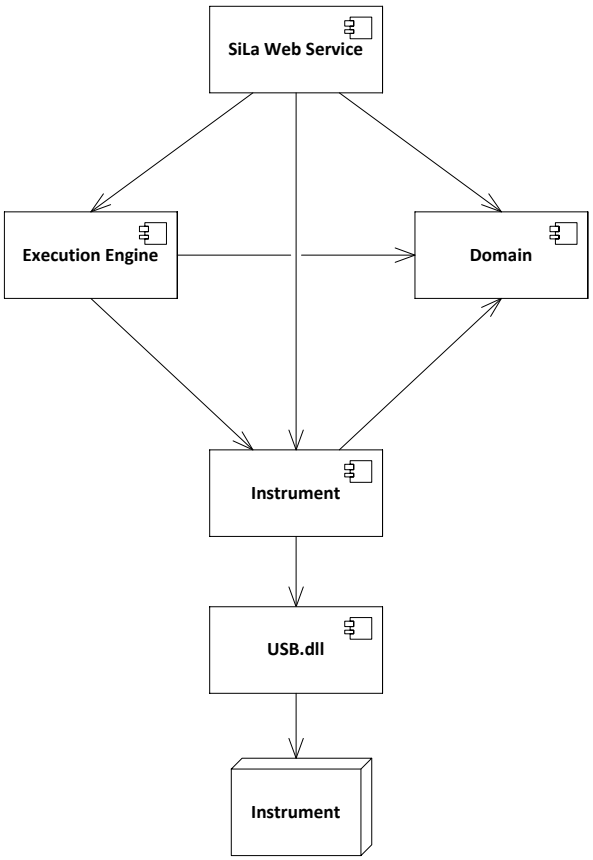


Figure 32. Selected modules with web service

Figure 32 shows an example, where automation system would only need protocol execution functionality, so it uses execution engine, instrument and domain modules to get all the required functionality.

3.4 Back-End and Front-End Communication

3.4.1 Façade

The user interface has only one access point to the back end and it's through the facade. This design forces developers to follow the chosen architecture, as they cannot have direct access to the back end. All functions in the façade are asynchronous and return values are returned with events. This way the developers cannot make blocking code accidentally. Figure 33 shows the user interface and the back end communication.

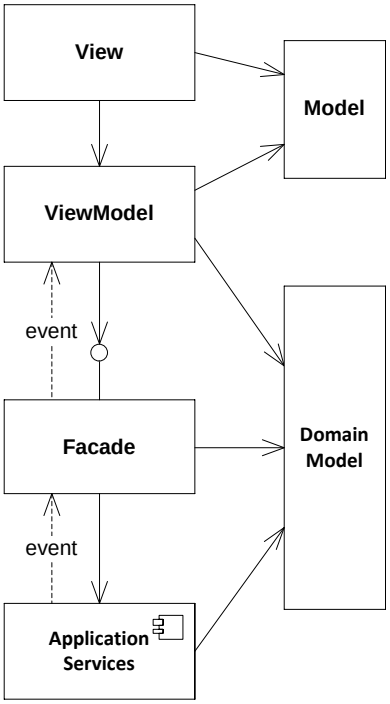


Figure 33. User interface and back end shared models

The user interface and the back end have both their own models. This prevents a situation when changing the model in the background thread would change the data in the UI (and vice versa). Both the user interface and the back end have their own event aggregators for internal communication. This prevents distributing too much information to outsiders, who do not actually need that information.

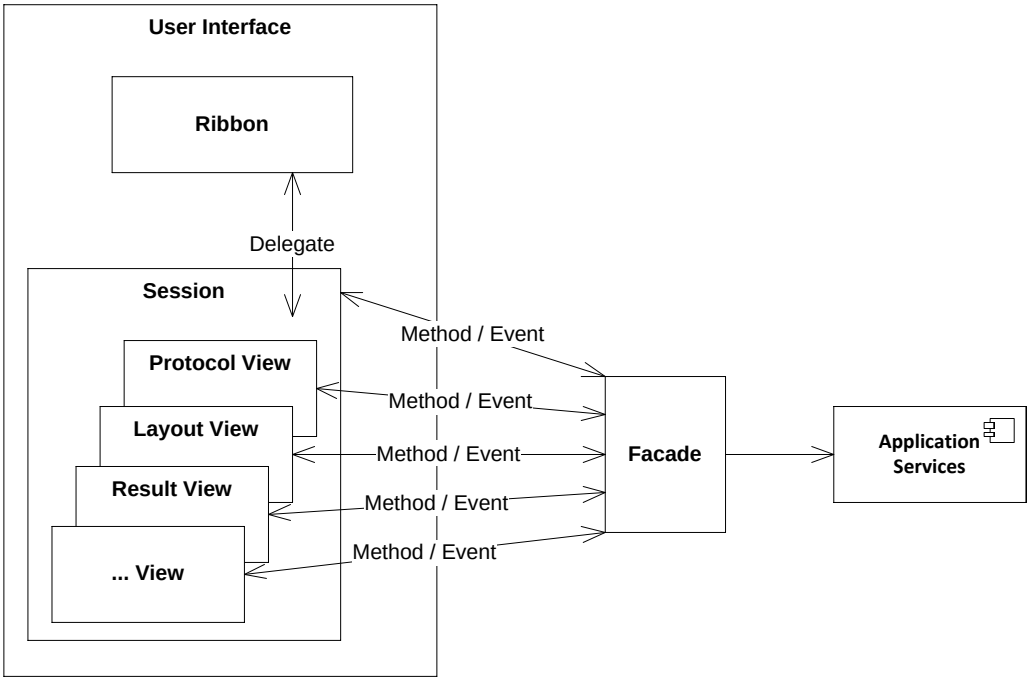


Figure 34. User interface and façade communication

All views inside the session are connected to the same façade, so all the views listen to the same events. So when one view requests for an update, all the views that are connected to the same façade will receive new data. This communication is illustrated in figure 34. In this way the data is not unsynchronized between views that show the data from the same session.

3.4.2 Asynchronous Method Execution

When a method from the façade is called, the façade will create a new task and execute the corresponding method from the service. The façade will get updates from the service with events and will pass these updates to the user interface with its own events.

Asynchronous method execution has the following steps:

1. User Interface listens to the façade's DataReady event.
2. User Interface executes the façade's GetData method.
3. The façade starts a new task that calls service layer.
4. The service layer gets data from the data storage and processes it to the correct format.
5. The service layer sends progress events with events.

6. Façade maps received data to models that the user interface uses.
7. Façade processes DataReady event.
    a. If the session is active, the façade sends DataReady event with new data
    b. If the session is not active, event is saved to queue
8. The user interface receives the DataReady event and updates controls with the new data

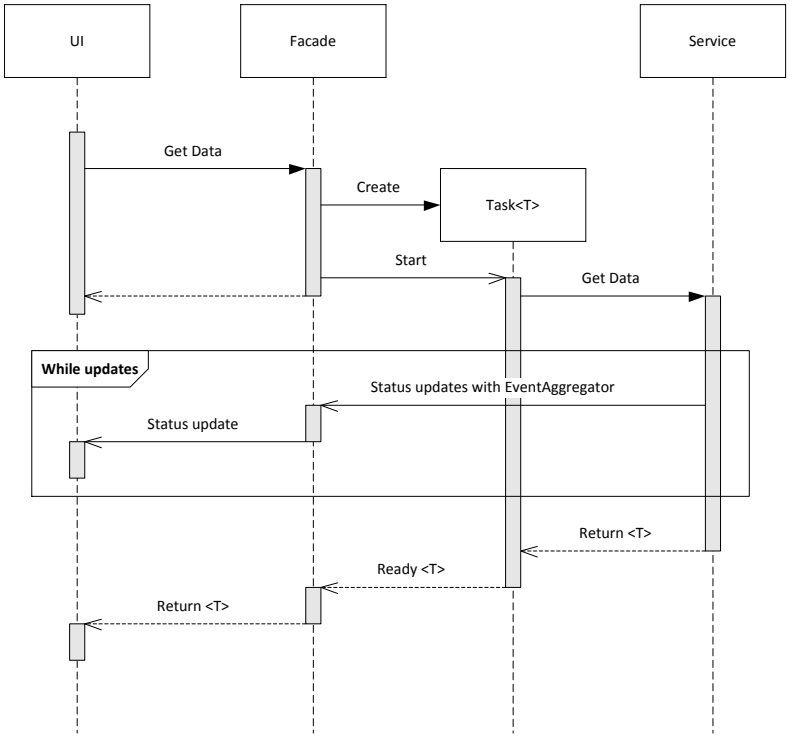


Figure 35. Common UI and back end communication situation

Every task should be able to be cancelled in the middle of the execution in case the user makes an action that would not require the previous action any more. This is illustrated in a sequence diagram in figure 36. When a task is cancelled, a cancellation token is set to false. When the task is finished, the state of the token is checked, and if the token is cancelled, then the response is ignored.

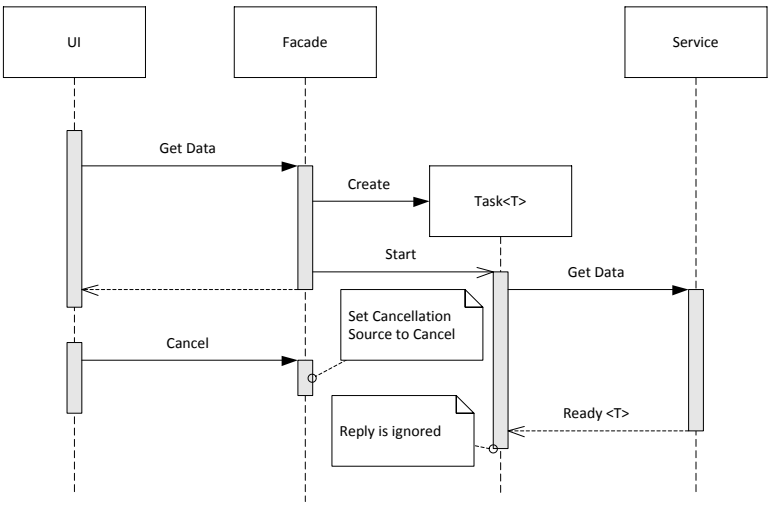


Figure 36. Cancellation of a task

If a task is known to be a long running task (execution of the task takes a long time), then a cancellation is sent to the back end with a cancellation method, as shown in the sequence diagram in figure 37.

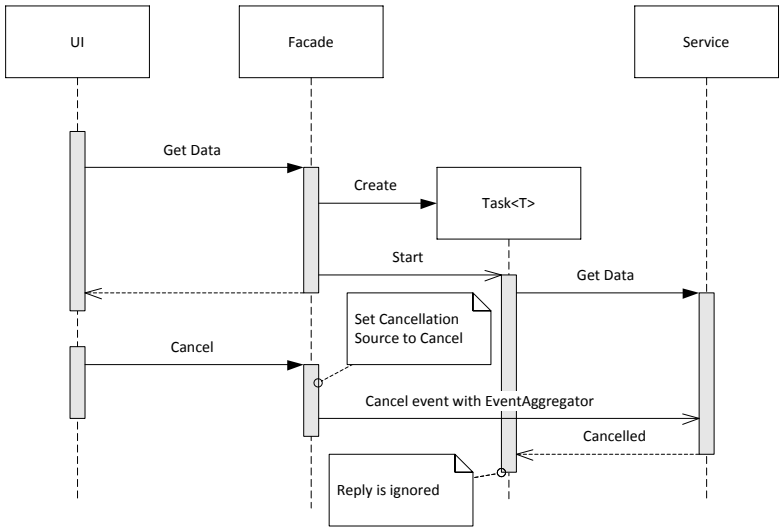


Figure 37. Cancellation of a long-running task.

Short tasks can finish their execution, but the return value is not passed on to the user interface. Processing of unnecessary data in the user interface would only slow down the application.

3.5 Back-End and Instrument Communication

Communication channels to the instrument are encapsulated and easily changeable. The instrument class uses the interface IInstrumentCommunication for communication. Communication classes implement this interface. Most of the instruments connect to a PC with USB.

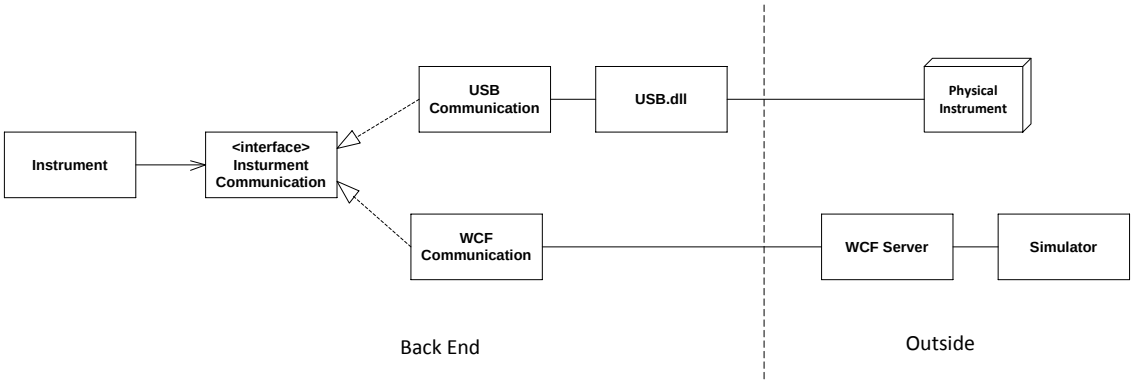


Figure 38. Physical instrument and simulator communication

By having this kind of encapsulation, the application is not aware of what kind of communication channel is being used for communication. For example, using a simulator instead of a real instrument does not require any code changes, and a decision between the simulator and the real instrument can be made on run time. This is illustrated in figure 38.

3.6 Back-End and Instrument Command Execution

3.6.1 Protocol Execution

A protocol is a set of command steps that specify what the laboratory instrument will do. Each step in the protocol may contain one or multiple instrument commands. Depending on the parent steps of the protocol steps, child steps might be executed multiple times.

One instrument command may have one or multiple responses from the instrument. For example measurement commands may have from one to an infinite number of responses and some status commands have only one response. Responses arrive from the instrument at intervals starting from a few milliseconds.

Protocol execution requires three modules from the back end. These modules are Execution Engine, Instrument and Domain. Figure 39 has classes and modules and their connections visualized. Some of the classes have functionality and some contain only data.

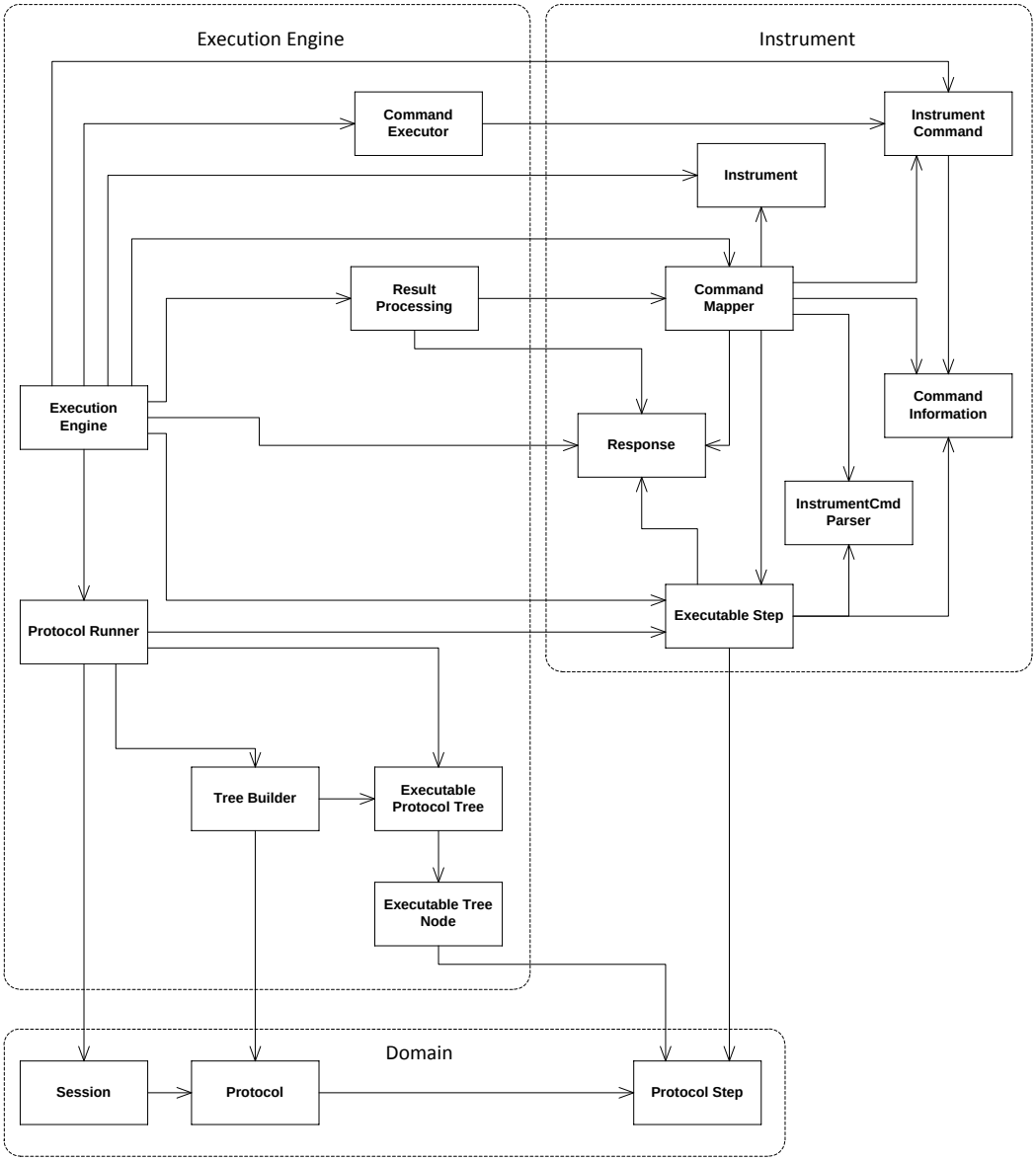


Figure 39. Protocol execution related classes

Each class is aware of only the classes' interfaces, not of concrete implementations. Concrete classes are used in figure 39 for the sake of simplicity. Some classes are on the platform level and some on the instrument-specific level. For example, the execution engine and command executor are implemented on the platform level. There is only abstract protocol runner on the platform level and concrete implementations on the instrument-specific level.

3.6.2 Command Execution

The command executor is responsible for executing new commands in its own thread. It has a concurrent queue and it loops through that queue as long as it has new command items. If some commands are blocking, then the executor waits until it is given the permission to continue. The result processor takes response strings to a queue and processes those in its own thread. Figure 40 shows modules and classes used in command execution.

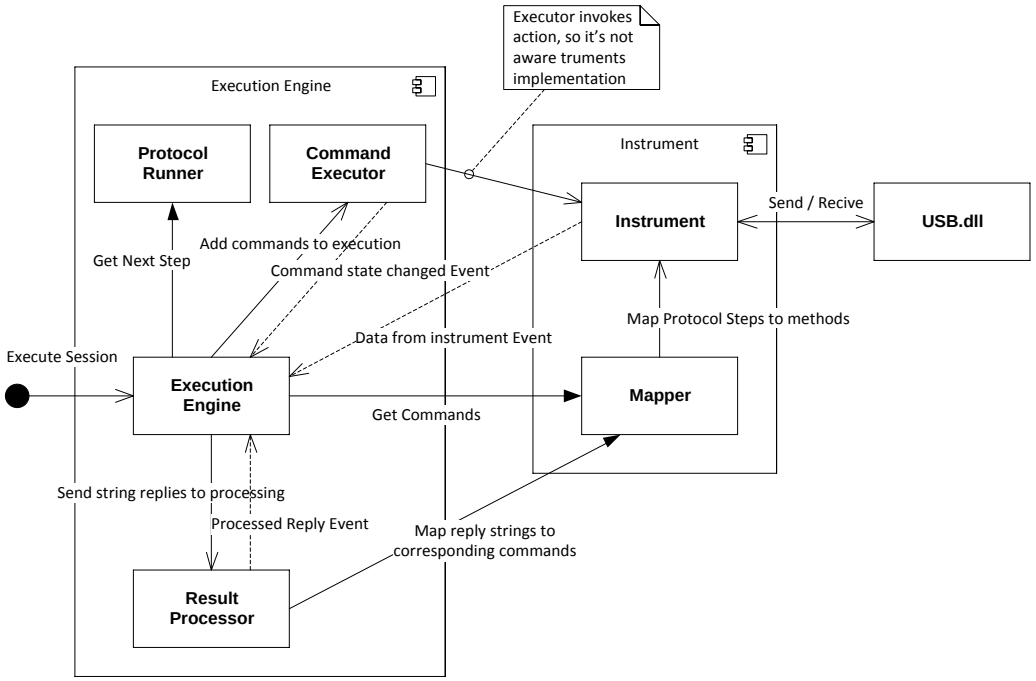


Figure 40. Command execution pipeline

Command execution runs on three different threads. The execution engine invokes a new data event asynchronously, so that the event is also processed in its own thread. Figure 41 illustrates the command execution activity as a diagram.

Command execution threads have following functions:

1. The thread is executing commands in the command executor's queue.
2. The thread is listening replies from the instrument.
3. The thread is parsing responses in the response handler's queue and the same thread also handles response in the execution engine. This ensures that responses are parsed and handled in order.
4. The thread (or any number of threads) notifies the user interface and passes data to persistor-classes, which saves the data in the database in their own threads.

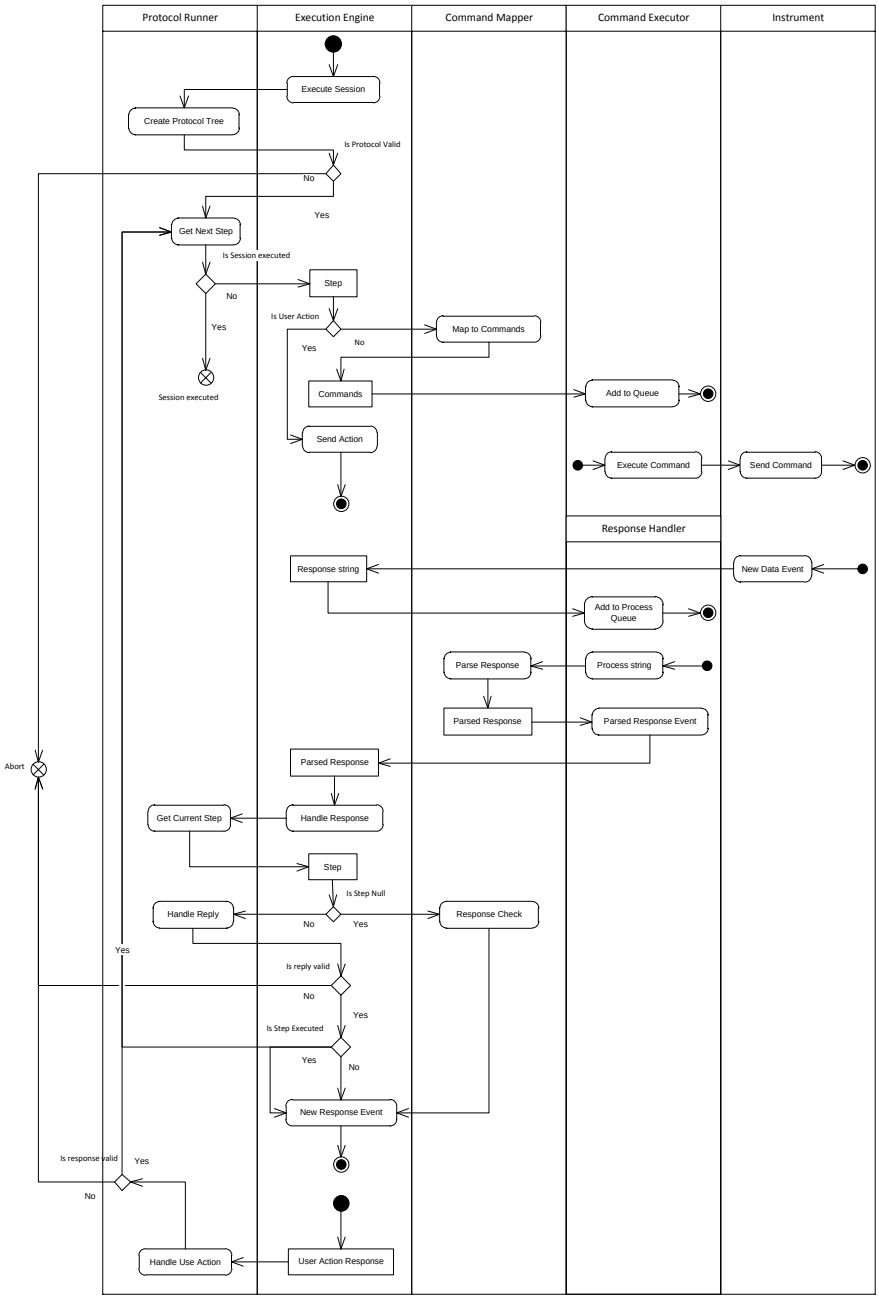


Figure 41. Command execution activity diagram

This kind of multi thread support makes the response processing non-blocking. This is extremely important when multiple instruments are connected to the same computer, as it is possible that responses arrive at very short intervals. It is not that crucial when the application finally handles the response and saves the data in data storage, but the application must be able to receive instrument responses all the time, so that it will not block instrument execution.

3.6.3 Response Handling

The response passes through multiple classes, each of them having a single responsibility. Figure 42 shows classes that take part in handling the response. Some of the classes have actual functionality related to response handling and some act only as a middle-man, meaning that they just pass the response onwards.

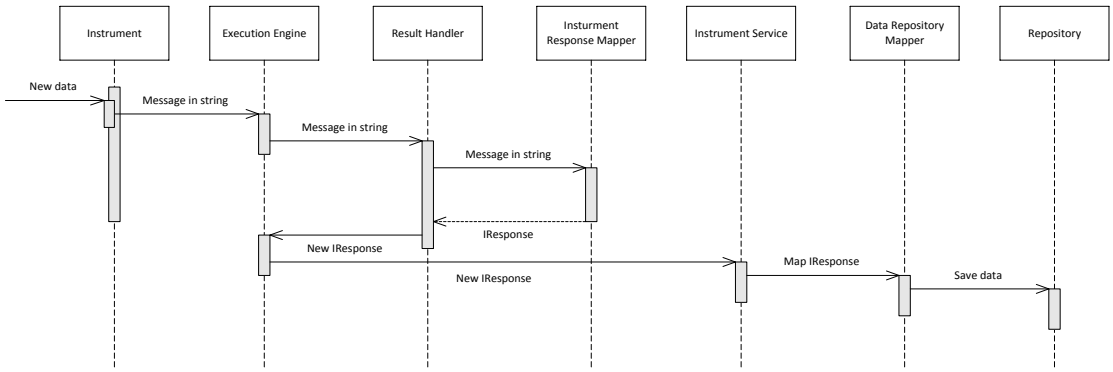


Figure 42. Message from instrument to database

Each data processing is done in its own thread. In this way the data from the instrument can be received without having to wait for processing and saving. Figure 43 shows a more detailed sequence diagram of the whole data processing activity.

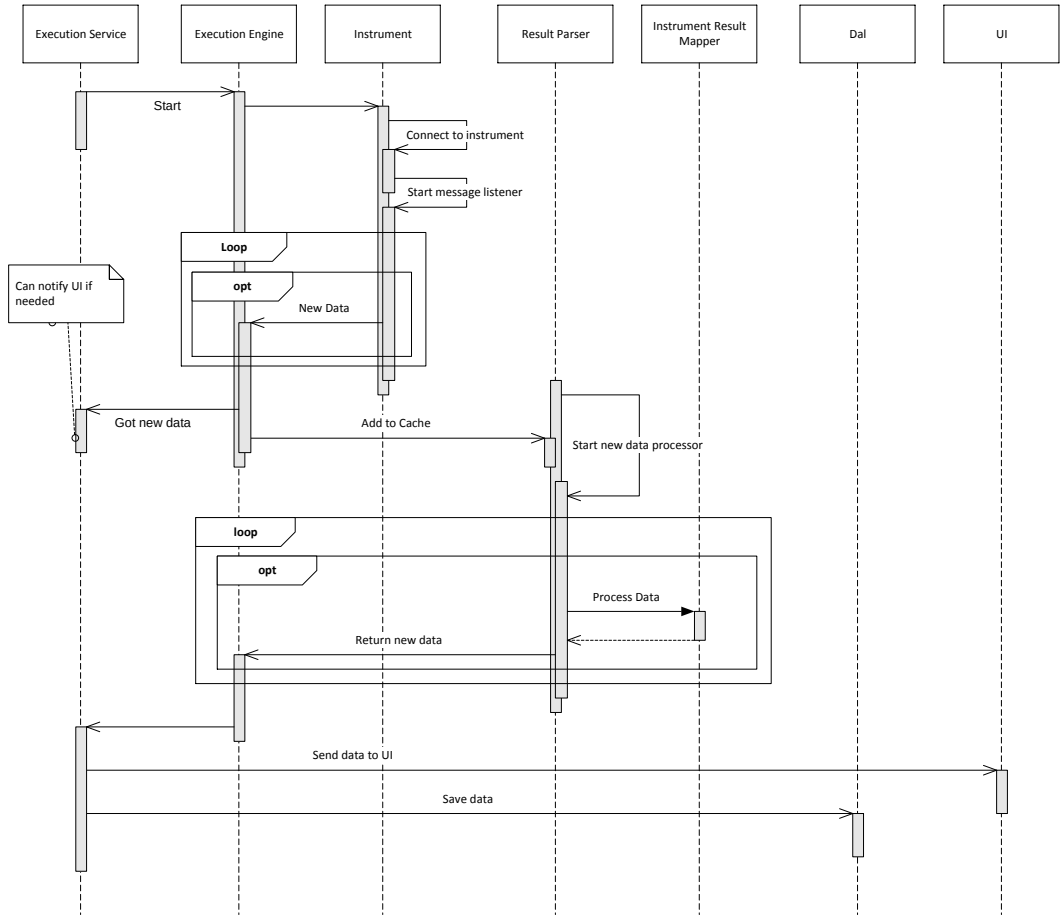


Figure 43. Response route

Response handling has the following steps:

1. Instrument sends a new data event.
2. Execution engine receives event and sends received message to the result parser.
3. Result parser queues message to cache.
4. Result parser inspects the queue all the time and parses new messages.
5. Queued message is parsed to instrument command.
6. New data is returned to execution engine with DataReady event.
7. Execution engine notifies with DataReady event.
8. Data is passed to DB to be saved.
9. UI notification is sent.

Saving data in the data access layer is done in the data persistor. The data persistor has a concurrent queue where all threads add the data that needs to be saved. The data persistor takes a batch of items from the queue and saves it in a database. It continues this operation as long as there is data to be saved. When new data is added to

the queue and the saving process is not running, it will be restarted. In this way, there are no unnecessary threads waiting for the data in the background.

3.7 Data Storage

3.7.1 Database

The database is designed to work with the Entity Framework using POCOs. When the Entity Framework is used, not all good database design practices can be followed. For example it is recommended that composite keys are not used, and Entity Framework prefers using unique identifiers as keys. The database can be roughly divided into seven different areas, as shown in figure 44.

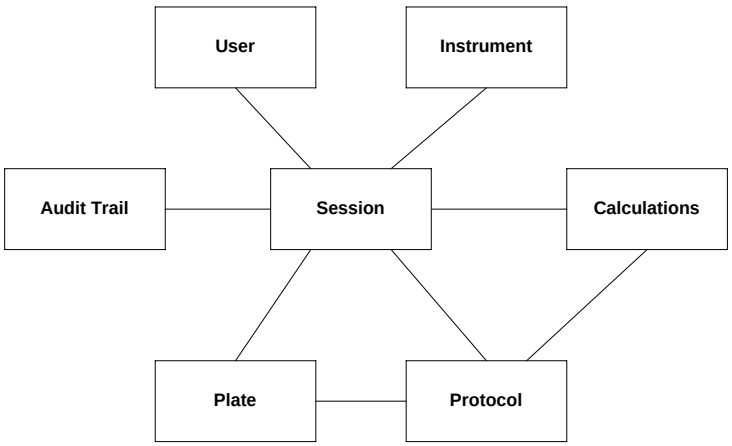


Figure 44. Database divided into areas

The database is designed using Table per Type (TPT) inheritance and Table per Hierarchy (TPH) inheritance. With TPT, each class has its own table and with TPH, the classes share one table and an extra column is added to the table, which defines the type of that row. The number of tables with TPH inheritance is kept at a minimum, because, for example if only one table would be used for all of the protocol steps, that table would have over 100 columns and most of the types use only under 10 columns. The benefit of TPH is much better performance and a smaller number of tables. Figure 45 shows some selected tables and what kind of inheritance is used in them.

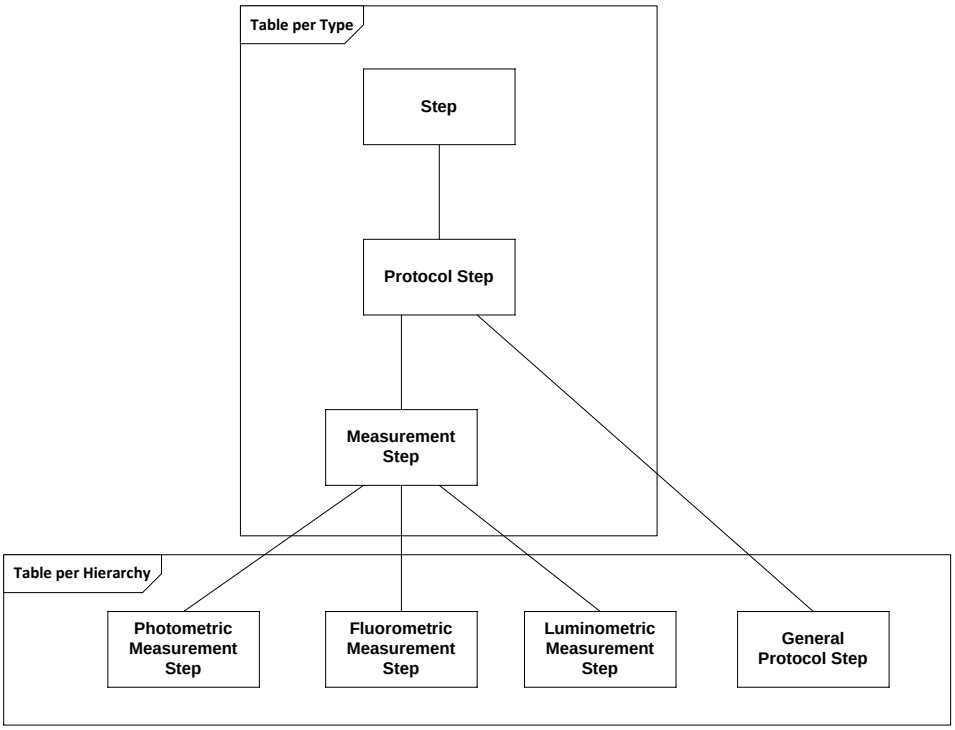


Figure 45. Table per Type and Table per Hierarchy inheritance

Figure 46 shows parts of the layout and the protocol areas and how their tables are linked together. In total the database has approximately 55 tables, and most of them use TPT inheritance.

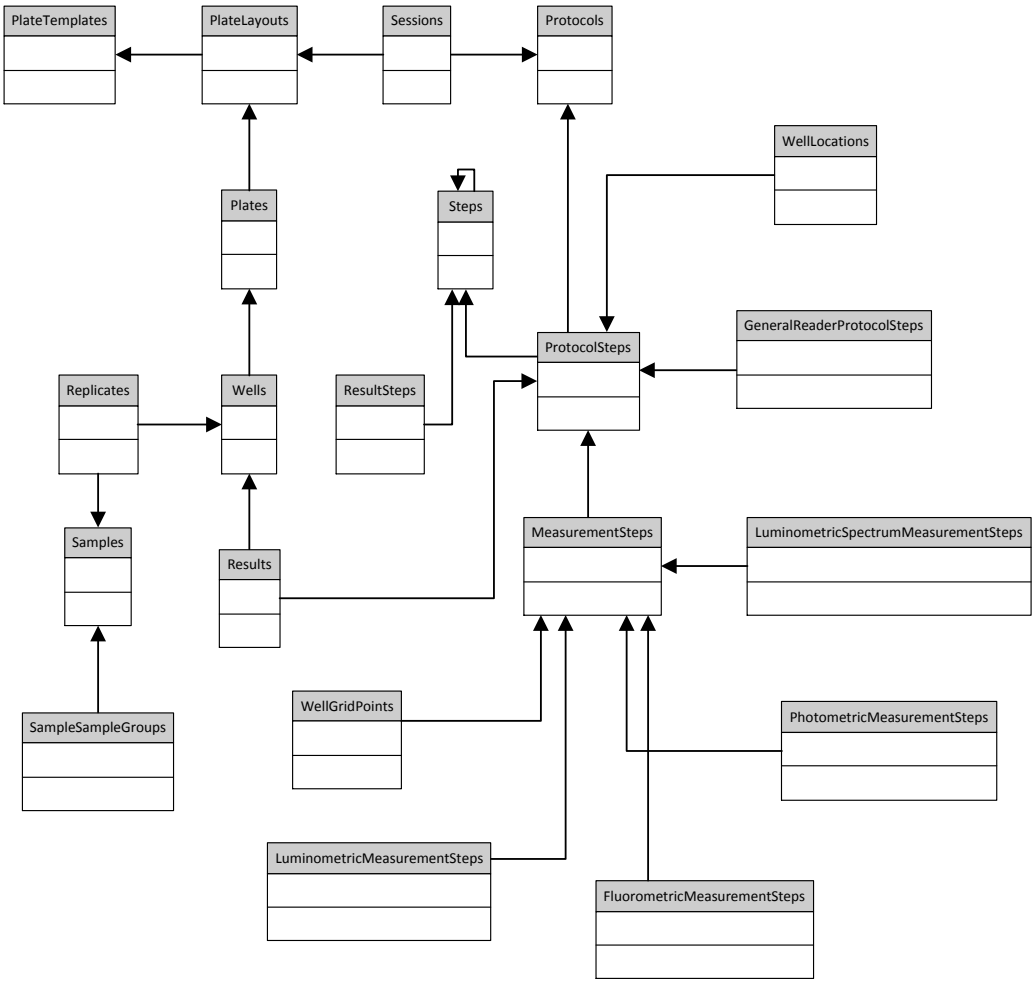


Figure 46. Plate layout and protocol steps table inheritance (only a few selected tables)

The database is created with the SQL script and POCOs are maintained manually. In this way, a correct database structure can be ensured (rather than let Entity Framework to create the DB). Because POCOs are not Entity Framework related, a change away from Entity Framework can be done if needed.

The database uses globally unique identifiers (GUID) as keys. Besides being a recommendation from Entity Framework, it gives one benefit. Using GUIDs ensures that the IDs will not collide even when data is inserted from multiple threads at the same time. GUIDs also make data import to an existing database easier, as there is no need to create new IDs for existing data.

Since the database is large and it has plenty of inheritance and links, the database context has to be split into smaller parts. For example, the context used for saving has only reference to Step, Protocol Step, Well and Result tables. This improves the per-

formance of the Entity Framework. Other performance improvement is gained by fetching the data using views, rather than fetching the data from original tables. Using views is only possible when the data is only shown to the user, but not when the data needs to be updated.

The database is designed to be used with LocalDB, which is management-free edition of Microsoft SQL Server Express. The database also works with Microsoft SQL Express and with full Microsoft SQL Server.

3.7.2 Repositories

The Entity Framework's DBContext is abstracted with the ContextAdapter. In this way it can be exchanged to any other data storage technology. Each instrument and the platform have its own Context. Some Entities have their own repositories and some are divided per object graph to keep the number of repositories lower.

A Context is always disconnected after data is loaded, so lazy loading is not used. Each repository has *included methods* that can define which children will be loaded. If the data is updated, the entities must be attached to the context.

3.8 Security

The application uses Windows authentication for user authentication. In normal use, login is not required when the application starts as the user has already logged into the operation system with a valid password. There is also a version of the application that will require the user to login in when the application is started.

When a user logs in (or starts the application), a new rights package is made from the user's group rights. This package is passed to the current thread. When services perform user authorization, they get the current user from the thread and do checks using a security component. The rights package is only updated when the user restarts the application.

4 Namespaces and Solutions

4.1 Namespaces

A namespace lets the user to organize the code and gives him or her a way to create globally unique types. They also provide assistance in avoiding name clashes between two sets of code.

Microsoft naming guideline recommends the following: [21]

CompanyName.TechnologyName[.Feature][.Design]

As the platform is developed with a single technology (Microsoft .NET) and it can easily be divided into its own parts, it uses a little different naming.

Company.Product.Family.[.Instrument].ModuleName

The naming guideline in this architecture strictly follows software project structure, so it is always clear in which project the namespace is.

4.2 Solutions and Project Structure

Each instrument and the platform have its own Visual Studio solution for the back end. Instrument solutions also have platform projects and instrument family common projects included in their solutions. The front end has its own solution and it has the back end added to same solution as library references.

There is also a solution that has the front end and all back end projects (the platform and instruments). This kind of "Keep everything" solution is used for continuous integration and it is also used by developers when making API changes and refactoring.

All solutions can be built with one command. The required libraries and 3rd party components are also included in version control, so there is no need to install anything on the development machine.

To get the environment running on the development machine, these are the required steps:

1. Get the latest codes from version control.
2. Execute the database creation script (requires MSSQL Server 2012 Express).
3. Build solution.

When the solution is always ready to be built, developers will not need to use time to copy new library files and compile multiple solutions. This is especially useful when a developer comes back to do some work with a project after being away from it for a while. It is quite frustrating to use hours to try to get the development environment to work again.

5 Selected Technologies and Patterns

5.1 Technologies

This chapter provides a brief description of the technologies used with the application. All the technologies were decided to be chosen from Microsoft .NET stack. This ensures good support for the products in the future.

.NET

.NET is a well supported and extremely versatile run-time environment. The application uses .NET Framework 4.5, which means that support for Windows XP was dropped out. Although several customers still use XP as their primary operating system, .NET Framework evolves at such a fast pace, that a decision to lock the application to an old technology could not be made.

Windows Presentation Foundation

The user interface was developed with Windows Presentation Foundation (WPF). WPF supports Model-View-ViewModel, which is also used by some of the JavaScript frameworks, such as. Knockout.js. WPF employs XAML to define user interface. XAML is also used as a definition language for Windows Modern UI applications. In reality, at the moment, there is no other option for desktop development in .NET stack, which is still developed further.

Entity Framework

Entity Framework (EF) is used for object relation mapping (ORM). Entity Framework supports POCOs (Plain Old CLR Object), so the domain does not have to be coupled to Entity Framework. Entity Framework's performance also reached a reasonable level with .NET 4.5. Another option would have been NHibernate, but at the moment the performance of these two is quite similar, so it was decided to choose the .NET stack option.

Windows Communication Foundation

Web Services were developed with Windows Communication Foundation (WCF). WCF might be a little heavy for the requirements of the application, but it will handle every situation that will be needed. WCF comes with the .NET stack and it is also used for inter-process communication.

Managed Extensibility Framework

Managed Extensibility Framework (MEF) was chosen to be the Dependency Injection container. MEF is a library that helps to create extendable applications and avoid hard dependencies. MEF comes with .NET Framework and does not require any additional libraries. It supports dynamic module loading, which is used in the user interface.

Event Aggregator

Event Aggregator is used for user interfaces internal messaging. The Event Aggregator service is a container for events, which helps to publish an event between decoupled modules. Event Aggregator is included in Microsoft Prism. Prism is a composite application library which helps to build and design rich-feature WPF applications.

SQL Server Express LocalDB

Microsoft SQL Server Express LocalDB is used for data storage. It does not require any configuration, only the library files installation on the target computer. This makes the application installation much easier, as there is no chance of SQL Server configuration errors. Also up scaling from LocalDB to full SQL Server is possible without any changes to the application.

3rd Party Technologies

The user interface uses 3rd party application libraries. These libraries provide better user interface controls than the .NET Framework, so although these libraries are not free of charge, it will save time from user interface development. Two common library providers are DevExpress and Telerik. As the components provided by these companies are fairly similar and developers had experience with DevExpress from previous products, decision to choose DevExpress was made based on that.

5.2 Patterns

Patterns are common approaches to problems which have been formalized and are often considered good programming practices. This chapter explains the most visible patterns used in this architecture.

5.2.1 Dependency Injection and Service Locator

Dependency injection and service locator take different approaches to achieve inversion of control. With dependency injection, dependencies are injected to components from outside. This behaviour is illustrated in figure 47.

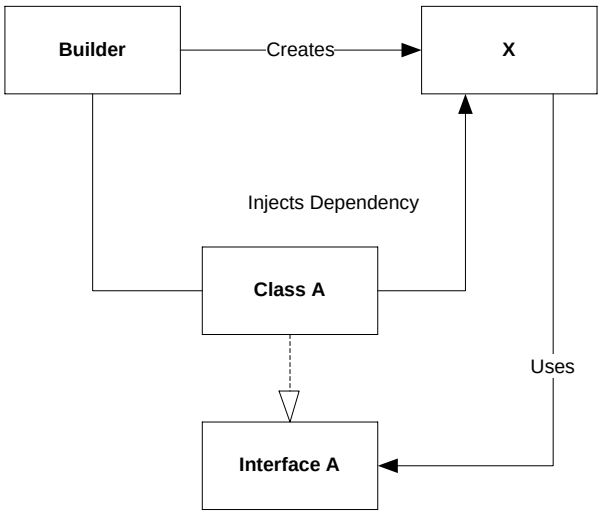


Figure 47. Dependency injection

With the service locator, the component uses the service locator to get the dependencies it needs. This is illustrated in figure 48.

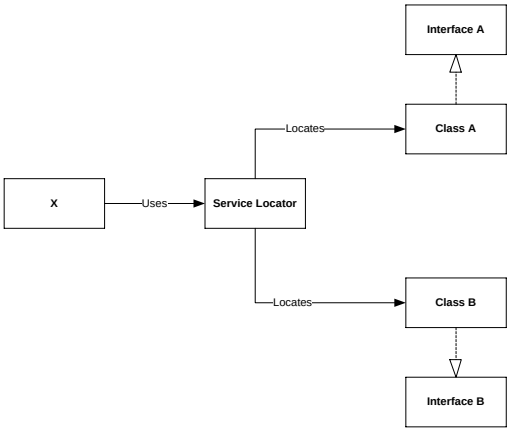


Figure 48. Service Locator

A benefit of the dependency injection is that the component is independent from the dependency mechanism, as with service locator, the component needs to be aware of the dependency mechanism. A benefit of the service locator is that the component does not need to advertise the dependencies it needs to external components. [17]

5.2.2 Model-View-ViewModel

Model-View-ViewModel (MVVM) is a presentation layer pattern, where the user interface (View) is separated from functionality (ViewModel) and data (Model). MVVM uses data binding to bind models and functionality to view. In this way view should not have any code in it. The separation and communication is shown in figure 49.

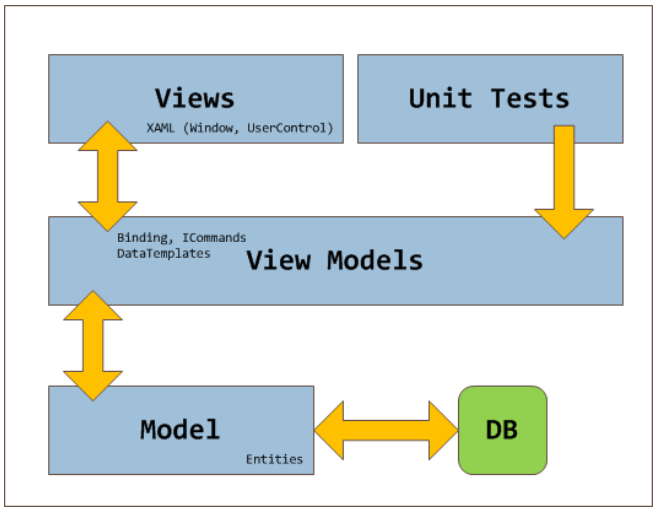


Figure 49. MVVM pattern [18]

A benefit of MVVM is separation. Designers can work on views, even when functionality on view models is not ready, by using the design time data, and developers can focus on creating functionality. View models are also unit-testable, so the presentation layer does not only have to rely on automated user interface tests.

5.2.3 Service Layer

A Service Layer defines an application boundary and its available operations from the perspective of the external components.

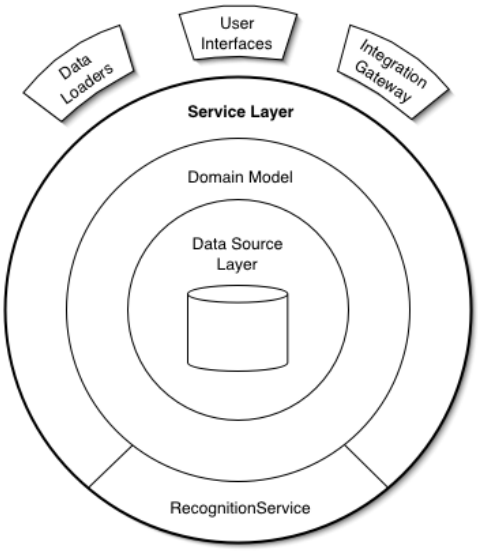


Figure 50. Service layer [7, 133]

The service layer interacts with the application logic and domain logic and it encapsulates the functionality of the application.

5.2.4 Domain Model

Domain modelling is the most important part of software design. Having a good model allows developers and a business to have a common language, which in turn makes the communication of requirements and the maintenance of the application much simpler. Also having a good model is a synonym for having a low representational gap, which means that the main concepts and their relationships from the real business model are represented almost identically in the domain model of the software. [23]

The domain model pattern allows one to design a domain model for the system in total freedom, without feeling a bound by platform and database constraints. The abstract domain model describes a number of processes and expresses some logic. This means that the domain model for an application should be the same for another application doing business in the same manner. [3]

5.2.5 Façade

A façade defines a higher-level interface for the subsystem. A façade shields the user from the complex details and simplifies the usage of the subsystem. It also decouples the subsystem from the user. Figure 51 illustrates the client connecting to a façade.

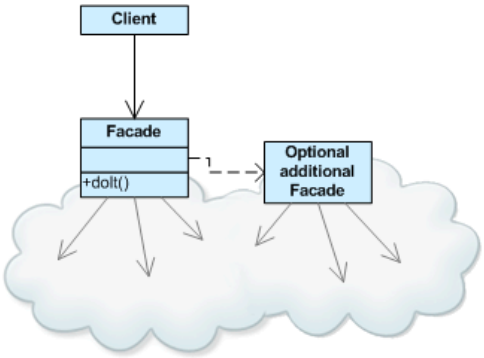


Figure 51. Client connects to Façade [16]

The facade object should be a fairly simple facilitator. It should not become an all-knowing “God” object, which is then an anti-pattern.

5.2.6 Repository

A repository mediates between the domain and data mapping layers. It uses a collection-like interface, which allows access to domain objects. Figure 52 shows the repository’s connection to the data source and business logic.

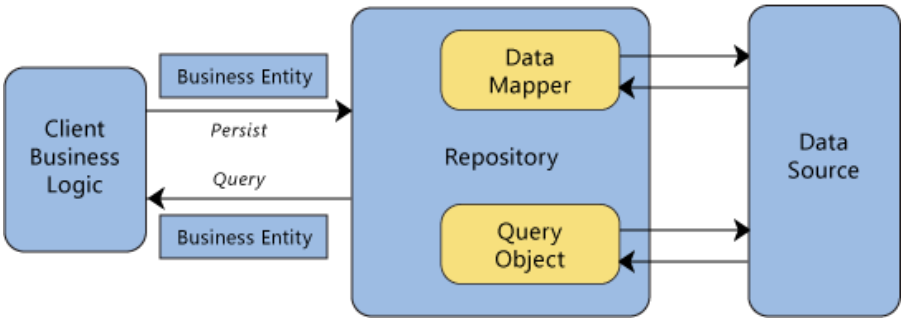


Figure 52. Repository [19]

A repository provides an object-oriented view of a data storage by encapsulating the stored set of objects and the operations that are performed over those objects. The repository provides a clean separation and one-way dependency between the domain and the data mapping layer.

5.2.7 Mapper

A mapper is an insulating layer between modules. It controls communications between two independent modules which are ignorant of each other.

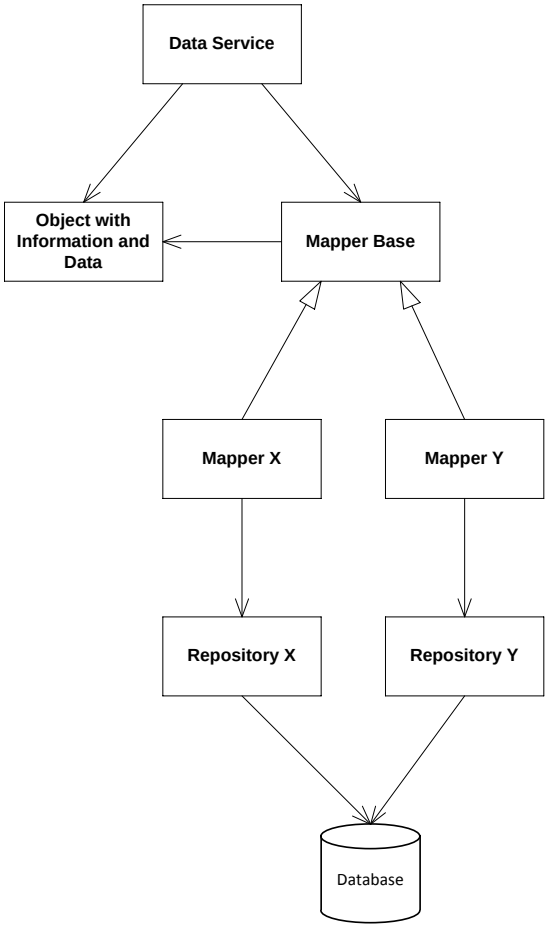


Figure 53. Mapper for storing data.

In figure 53, data service receives the data from two different providers, X and Y. Data service passes the data to the mapper linked to the provider (either Mapper X or Mapper Y). In this way the service does not have to know how to link the data correctly to

the repositories and the repository does not have to have knowledge of any other information besides the data.

5.3 Anti-Patterns

An anti-pattern is an opposite of a pattern, so it is often considered a bad programming practice. Two anti-patterns used in this architecture, are described here, as well as reasons for their use.

5.3.1 Anemic Domain Model

A domain model is an object model of the domain. The basic idea of the domain model is that the functionality is included in models. In an anemic domain model functionality is in a separate class and the model is just getters and setters for the data. This is usually considered an anti-pattern. [25]

PlateLayout used in this example, defines the specifications and content of a microplate, which is a plate with multiple wells as small test tubes.

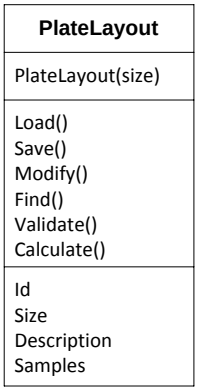


Figure 54. Model with data and functionality

In the full model, as shown in figure 54, PlateLayout-class has the functionality for create the object and persistence of the data. In figure 55 the factory and the repository patterns are added to the model, so the full model is separated into multiple classes.

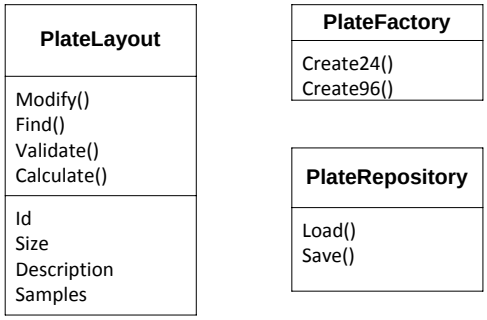


Figure 55. Model, Factory and Repository

By following the Single Responsibility Principle, PlateLayout is divided into a class with functionality and a class with a model. This is shown in figure 56. When this division is made, the model is an anemic domain model.

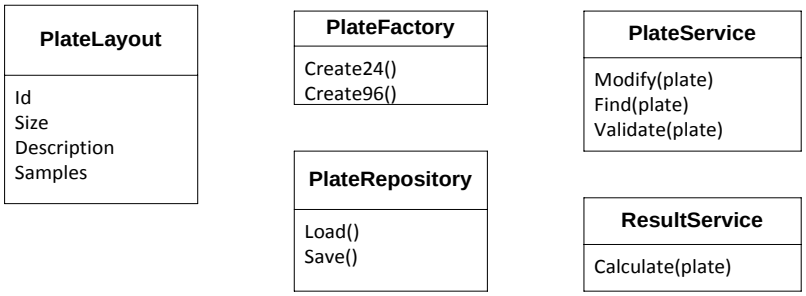


Figure 56. Model, Factory, Repository and Services

Having an anemic domain model in this kind of situation is completely acceptable. This makes the code understandable, testable and models can be used as a light weight data transfer objects. Dividing the functionality from a model into service classes is mainly a matter of taste.

5.3.2 Service Locator

The service locator can also be an anti-pattern. This is because when a class has the service locator's implementation, it can request anything it wants from the service locator. This kind of behaviour is similar to a situation when the application would have all dependencies configured to the service locator as global variables and global variables are an anti-pattern. The service locator also hides a class's dependencies, causing run-time errors instead of compile-time errors. It also makes the code more difficult to maintain because it becomes unclear when one would be introducing a breaking change.

[24] This does not mean that the service locator is automatically an anti-pattern, but there is an option that it can be used in a "wrong" way. It is recommended that dependencies should be injected through the constructor, rather than letting the class use the service locator as it wishes.

6 Conclusions

The aim of the project was to design architecture for a workstation software platform to control the connected laboratory instruments. The platform had to be modifiable, as it had to be able to support multiple different scenarios and different laboratory instruments, from which all are not yet known. It had to be maintainable, as it will have a long lifespan, during which it will have multiple modifications. At the stage when the document was written, the application with the architecture described here had been under development for a couple of man-years. As the architecture was designed in an iterative way, it has changed and will continue to change during the application's life cycle.

In general, designed decoupled and modular architecture fits the application well. It gives enough flexibility so the design can provide functionality for different use cases, and changes to the design will only affect small parts of the architecture. This kind of architecture allows the application to be used in multiple different ways, for example as a centralized service or as a stand-alone application, and the needed modules can be used with an automation service.

Platforms should not be considered as a universal solution for everything. If applications that platform should support are not related, instead of a platform, re-usable modules should be used and companies should have ready-made skeletons for different parts of the application.

The designed architecture may sometimes be too general, as when some decisions were made there was not enough knowledge of the final use cases. Although the architecture was aimed to be kept as simple as possible, having threading, abstraction and factories make it complex. It was still easy to develop even without having a full understanding of the whole architecture.

Finally, some main concerns and problems that arose during development are addressed here.

The performance of the application is and will be one of the main concerns. It is extremely important that an application is responsive all the time and a user gets a response for all the actions he/she makes. Of course, the application must perform these actions in a reasonable time, so only having responsiveness all the time is not enough.

To ensure this, the performance of the application is monitored continuously during the development, as the application has a habit of slowing down when more features are added. There are some automated tests that measure the performance, and they are executed with the continuous integration build, but primarily performance monitoring is done manually with the help of commercial applications.

Another big concern with the architecture was **instrument communication** as a new version of the application has to support multiple simultaneous instruments. Communication includes sending commands, handling responses and saving data into data storage. A concurrent model that was designed and implemented is working well, so handling responses from multiple instruments is not blocking any processing and it does not have too high CPU usage.

The Entity Framework brought various problems regarding configuration and data updating. The performance has been good enough as the latest version supports query caching. To cache most common queries, the application does queries in the background so when a user requests the data, the query will be fast. The data model in use has too much inheritance and child references and this causes problems with the Entity Framework. Unfortunately these problems came out too late, so at that point it was not worth changing it into another solution. If the decision could be made again, a more traditional ADO.NET (datasets and procedures / SQL-queries) approach instead of Entity Framework would be considered. The abstraction level of the data context would be moved to a repository. This would make the usage of the EF data context or any selected context easier. Currently it seems that the abstraction level is on an unnecessary low level.

Because many of the classes have background processing and environments are created with a dependency injection container, multiple checks needs to be done when disposing the objects when closing the session. DI container keeps references to created objects and will not allow classes that have background processes to be garbage collected automatically. As known, some customers keep this application on for months, and this is a potential place for **memory leaks**.

The decision to use the façade to unify asynchronous execution was good. In this way it is impossible to make blocking calls from the back end. The façade gives a good separation between the user interface and the back end. Now user interface develop-

ers do not have to have knowledge of the back end’s internal functionality, which makes development simpler.

In the end, everything seems to be going quite smoothly and there might be a happy ending for this platform. However that is something to be seen in a few years.

References

1 Plessel Todd. Data Model Project, Quality Assurance Plan [online]. Lockheed Martin / US EP; 2 November 2002.
URL: http://codecourse.sourceforge.net/materials/Quality-Assurance-Plan.html
Accessed: 10 July 2012

2 Csgillespie. CPU and GPR Trends over Time [online]. 25 January 2011.
URL: http://csgillespie.wordpress.com/tag/ggplot2/
Accessed: 18 July 2012

3 Glass Robert L. Facts and Fallacies of Software Engineering. Addison-Wesley; 2002.

4 Esposito Dino. Microsoft .NET Architecting Applications for the Enterprise. Microsoft Press; 2008.

5 Arking John, Millett Scott. Professional Enterprise .NET. Wrox; 2009.

6 Microsoft Spain. Domain Oriented N-Layered Application [online]. 21 November 2012.
URL: http://microsoftnlayerapp.codeplex.com/
Accessed: 18 July 2013

7 Fowler Martin. Patterns of Enterprise Application Architecture. Addison-Wesley Professional; 2002.

8 Martin Robert C. Clean Architecture [online]. 22 November 2011.
URL: http://blog.8thlight.com/uncle-bob/2011/11/22/Clean-Architecture.html
Accessed: 1 August 2012

9 Cockburn Alistair. Hexagonal Architecture [online]. 19 June 2008.
URL: http://alistair.cockburn.us/Hexagonal+architecture
Accessed: 1 August 2012

10 Martin Robert C. Clean Architecture presentation [online]. 7 June 2012.
URL: http://ndcoslo.oktaset.com/t-4868
Accessed: 1 August 2012

11 Albahari Joseph, Albahari Ben. C# 4.0 in a Nutshell. O'Reilly Media; 2010

12 Leonard Tom. Dragged Kicking and Screaming: Source Multicore [online]. Valve; 9.3.2007.
URL:http://www.valvesoftware.com/publications/2007/GDC2007_SourceMulticore.pdf
Accessed: 20 July 2012

13 Palermo Jeffrey. The Onion Architecture [online]. 29 July 2008.
URL: http://jeffreypalermo.com/blog/the-onion-architecture-part-1/
Accessed: 15 July 2012

14 Martin Robert C. Clean Architecture [online]. 13 August 2012.
URL: http://blog.8thlight.com/uncle-bob/2012/08/13/the-clean-architecture.html
Accessed: 22 October 2012

15 Martin Robert C. Screaming Architecture [online]. 30 September 2011.
URL: http://blog.8thlight.com/uncle-bob/2011/09/30/Screaming-Architecture.html
Accessed: 22 October 2012

16 Design Patterns [online]. Sourcemaking.com; 2009.
URL: http://sourcemaking.com/design_patterns
Accessed: 6 November 2013

17 Burns Kyle. Dependency Injection vs. Service Locator [online]. 27 April 2012.
URL: http://geekswithblogs.net/KyleBurns/archive/2012/04/27/dependency-injection-vs.-service-locator.aspx
Accessed: 6 November 2013

18 Shen HengBin. MVVM Pattern Simplified [online]. 2 July 2012.
URL: http://shenhengbin.wordpress.com/2012/07/02/mvvm-pattern-simplified/
Accessed: 6 November 2013

19 The Repository Pattern [online]. MSDN – Microsoft.
URL: http://msdn.microsoft.com/en-us/library/ff649690.aspx
Accessed: 12 November 2012

20 What is an Enterprise Application? [online] MSDN – Microsoft.
URL: http://msdn.microsoft.com/en-us/library/aa267045%28v=vs.60%29.aspx
Accessed: 20 November 2013

21 Namespace Naming Guidelines [online]. MSDN – Microsoft.
URL: http://msdn.microsoft.com/en-us/library/893ke618%28v=vs.71%29.aspx
Accessed: 16 August 2012

22 Creating N-Tier Application with C#. Pluralsight; 2012.
URL: http://pluralsight.com/training/Courses/TableOfContents/n-tier-apps-part1
Accessed: 5 January 2013

23 Gutierrez Alberto. How to create a good domain model. Top 10 advices [online]. 17 May 2010.
URL: http://www.makinggoodsoftware.com/2010/05/17/how-to-create-a-good-domain-model-top-10-advices/
Accessed: 5 January 2013

24 Gauffin Jonas. Service locator is not an anti pattern [online]. 9 December 2012.
URL: http://blog.gauffin.org/2012/09/service-locator-is-not-an-anti-pattern/
Accessed: 5 January 2013

25 Fowler Martin. Anemic Domain Model [online]. 25 Novermber 2003.
URL: http://www.martinfowler.com/bliki/AnemicDomainModel.html
Accessed: 7 April 2013

26 G. M. Amdahl. Validity of the Single-Processor Approach to Achieving Large Scale Computing Capabilities. In AFIPS Conference Proceedings, pages 483–485, 1967.

27 Amhdal's Law, Wikipedia.
URL: http://en.wikipedia.org/wiki/File:AmdahlsLaw.svg
Accessed: 7 April 2013

28 Martin Robert C. Design Principles and Design Patterns.
www.objectmentors.com. 2000.
URL:http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf

29 Martin Robert C. Agile Principles, Patterns, and Practices in C#, Prentice Hall. 20 July 2006.

30 Professional Enterprise .NET [online] 10 Jan 2011
URL: http://proent.codeplex.com/
Accessed: 7 April 2013